The Design & Implementation of Modular Smart Dust

Patrick Pannuto

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-208
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-208.html

December 17, 2020

The Design & Implementation of Modular Smart Dust

by

Patrick William Pannuto

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Prabal K. Dutta, Chair
Professor Kristofer S. J. Pister
Professor David E. Culler
Taisei Professor Khalid M. Mosalam

Fall 2020

The Design & Implementation of Modular Smart Dust

Abstract

The Design & Implementation of Modular Smart Dust

by

Patrick William Pannuto

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Science

University of California, Berkeley

Associate Professor Prabal K. Dutta, Chair

Advancements across the computing stack now allow the realization of self-contained, sensing, computing, and communication devices in less than a cubic millimeter of volume—Smart Dust. Today's millimeter-scale systems are often one-offs, however. They are built to prove they can be built or for one particular application, but they are not yet so numerous and diverse to satisfy the vision of ubiquitous, ambient intelligence. This work aims to identify and address challenges in the move from tens to tens of millions of Smart Dust computing systems.

The core contribution of this dissertation is bringing modularity to the millimeter-scale computing class. Much of the rich panoply of conventional systems is driven not by the diversity of individual chips alone but by the number of different ways in which they can be synthesized into novel designs. Composition-oriented design of systems is enabled in part by the interconnect technologies that lie between components. This dissertation identifies the limitations of extant interconnect technologies for the millimeter-scale computing class and then designs, implements, and evaluates a new interconnect bus and interconnect protocol that enables composable Smart Dust systems. As a final contribution, this dissertation shows how this same interconnect can be leveraged to support key operations for bringup and deployment of Smart Dust systems, in particular programming and debugging.

Updates and errata to this dissertation can be found at: patpannuto.com/dissertation

To all the Dreamers and Builders:

Go make something cool.

Contents

List of Figures

List of Tables

Acknowledgments

To Family, John, Linda, Steve, and Julie Pannuto: It is an incredible and in some ways indescribable privilege to have and to know that you have a backstop of an endless well of unquestioning love and support. Thank you for your infinite patience, especially in the many times in this process that I disappeared without notice, and for the steadfast help and support every time I came back. I cannot imagine life without you.

To Friends: To Haven Moore, who somehow found himself by my side (and cleaning my kitchen) in both the first and last days of this adventure, as well as so many in between—thank you for so often knowing what I needed, even when I did not. To Lauren Leader, for patience, love, and support, for someday teaching me how to get things done in the right order, and for so much more of our life yet to come. And to Bill Grier, Jill Grier, James Huebschman, Jacob Wimmer, Erin Wimmer, Brian Roberts, Tyler Sanderson, John Krzemien, and Shazil Naqvi, for being some of the best friends anyone could ever possibly ask for—here's to a lifetime together.

To the People and Friends of Lab11: In many ways, a Ph.D. is defined by the people you spend it with, and I could not be more grateful to have spent my time with all of you. Thank you to Brad Campbell, Zakir Durumeric, Noah Klugman, and David Adrian, my partners in every part of graduate life that does not make it into a dissertation. Thank you to all those in Lab11, those who came early and helped to build an incredible institution, especially Thomas Schmid and Aaron Shalev, those who I spent so many years with, Ye-Sheng Kuo, Ben Kempke, Sam DeBruin, Will Huang, Meghan Clark, Branden Ghena, Rohit Ramesh, and Thomas Zachariah, those to whom we pass on the legacy, Joshua Adkins, Neal Jackson, Matt Podolsky, Jean-Luc Watson, Shishir Patil, and all those yet to come.

To the Michigan Micro Mote team: What an incredible thing to have built. Thank you for letting me share in the journey and contribute a small part of it. Thank you especially to ZhiYoong Foo, Yoonmyung Lee, and Gyouho Kim for all of the long and late hours getting things working, your patience and lessons in all of the electrical engineering that I did not know, and for being my friend. And thank you to David Blaauw, for meeting an eager undergraduate at a Borders over Christmas, for later believing in and betting on their ideas in a whole lot of silicon, and for the staunch support as my career began to bloom.

To my Teachers and Mentors along the way: I think you are allowed some indulgences in the acknowledgements section of a dissertation. I remember nearly every teacher I have ever had, from Kindergarten through my final graduate course, in what was perhaps the 24th grade. I suspect I think of at least one former teacher nearly every day of my life, for any number of reasons. I am too afraid of regretting an omission to try list every one here, but I would like to try to thank many of the people who have made me who I am today:

I am a teacher because of my second grade teacher, Mrs. Schroeder.

I am in STEM because of my third grade teacher, Mrs. Smith.

I am a much better teacher because of my fifth grade teacher, Mrs. Boswell.
I am a computer scientist because of my ninth grade math teacher, Mr. Lamkin.
I am an engineer becuase of my *FIRST* mentors, Mr. Hildebrandt and Paul Slaby.
I think in systems because of Mr. Glenn.
I travel the world because of M^{me} Wieten and Frau Barner.
I am still in STEM because of Mrs. Deyo and Mr. Domanski.
I am a researcher because of Dr. Manish Karir.
I am a professor because of Professors Mark Brehob and Peter Chen.
I am an electrical engineer because of Dr. David Blaauw.
I am an ethical hacker because of Dr. J. Alex Halderman.
I am a Dr. because of Dr. Prabal Dutta.
I am immensely grateful to everyone who has a been a part of shaping my life. I can only hope to pay forward what I have learned from each and every person, and in so doing further amplify the impact of your wisdom and kindness.

To my Committee, Kris Pister, David Culler, Khalid Mosalam, and Prabal Dutta: I find somehow that even when I do not intend to, I end up on the path less traveled in life. Thank you for being exactly the mentors and guides that I needed, exactly when and how I needed them. You have set the shape of my career, and I hope to live up to the models you exemplify.

To my Advisor, Prabal Dutta: What a wild thing, to take a chance on the kid who was falling asleep in your class. To this day, I wonder what you saw, and when the opportunity presents itself, I hope that I will see it in someone too. Thank you for teaching me how to think. Thank you for trusting me with an interesting problem—a dissertation-worthy task!—from day one, and for giving me the freedom and flexibility to explore so many other things along the way. Thank you for backing me, visibly and invisibly, all the times that it really mattered. My time in graduate school has been an absolute joy, thank you for making that possible.

Chapter 1

Introduction

This dissertation is about modular design and systems considerations for the constrained computing platforms known as Smart Dust. The distinguishing features of Smart Dust devices are that they are both computationally interesting and physically small. Computationally interesting means a device is in some way intelligent: it senses data about its ambient environment; it supports arbitrary computation; it offers communication capability. Physically small for the purposes of this dissertation means a device with a principle node dimension on the order of a millimeter. Conceived of in the 1990s [1], it took until 2011 to realize the first fully self-contained device in the cubic millimeter envelope [2]. Today, the question is no longer how to build any one cubic millimeter computer but rather how to build ten million of them and how to make them useful. Hence this discussion of modular design. This dissertation itself is one piece of a much larger whole. The scope of the Michigan Micro Mote project encompasses several theses [3–7] and dozens of papers and talks [2, 8–17]. These concepts are the product of an enormous effort by a diverse and talented team.

The core contribution of this dissertation is the systems reasoning for an extensible, modular design and the corollary architectural insights that enable modular Smart Dust chips to be deployed ad-hoc. The Smart Dust in Figure 1.1 is set apart by being the first such devices made entirely of modular, reusable components. Roughly half of the individual chips are shared across all three designs.

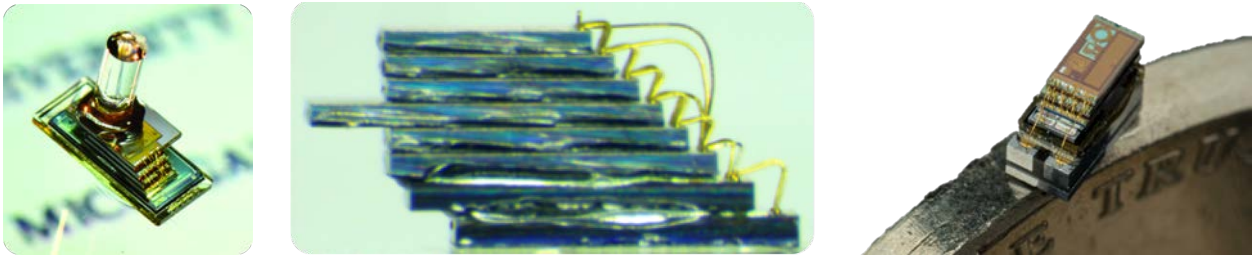


Figure 1.1: **Examples of Smart Dust systems.** Three millimeter-scale sensing systems composed using the modularity and system design principles outlined in this dissertation.

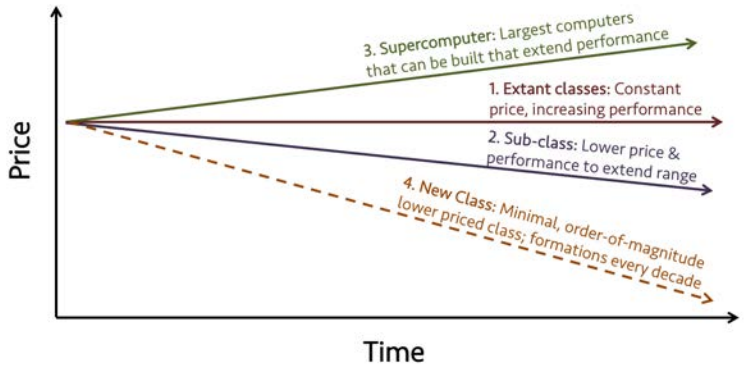


Figure 1.2: **Bell's Law for the birth and death of computer classes.** This figure is a reproduction of the trends identified by Gordon Bell [18, 19]. Summarized, Bell observes that steady year-over-year technological advancements allow for (1) improved performance at fixed cost, (2) lower cost for fixed performance, or (3) greater overall capability. Roughly every decade, however, an amalgamation of advancements come together to allow for (4) a greater-than-iterative change that result in the creation of a new "class" of computing.

1.1 Classes of Computing

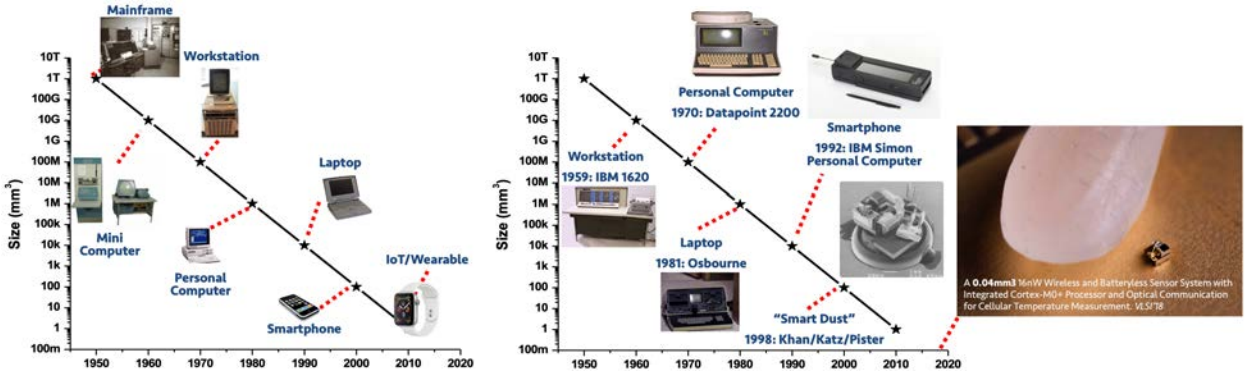
The emergence of Smart Dust is not the first time that a new and different-looking type of computer has come into the world. In 1975, Ken Olsen and Gordon and Gwen Bell founded The Computer Museum, a precursor to today's Computer History Museum [20]. As part of this process, Gordon Bell began to use the look back as a way to look forward and coined *Bell's Law for Birth and Death of Computer Classes* [18, 19]. Bell's law, reproduced in Figure 1.2, groups computational systems into classes based on macroeconomic trends:

> A computer class is a set of computers in a particular price range with unique or similar programming environments ([...]) that support a variety of applications that communicate with people and/or other systems. (Gordon Bell [18, p. 86])

This grouping into classes shows how progress in underlying technologies manifests in computing devices. In the normal steady-state, extant classes improve iteratively over time. Approximately every decade, however, the collection of many advancements that allow systems to 'do more using less' allow for the imagining of a new "minimal" class, computing systems that are an order of magnitude less expensive but are also as or more capable than their equivalent from the decade before.

This decennial trend is not restricted to economics, however. Figure 1.3 presents a new corollary to Bell's law based on physical size. At approximately the same cost, every decade one can buy a general purpose computing device that is an order of magnitude smaller. Looking further, while Figure 1.3a looks at the widespread availability of each physical computing class, Figure 1.3b tries to find the first instances of devices in each class. In each case, a new widespread physical class is preceded by a first instance in that class around ten

years earlier. This suggests that there are perhaps two watersheds required to promulgate a new physical class of computing: the first is the collection of advancements that enable the first instance of a class, the 'proof-it-is-possible'; the second is the collection of advancements required for the new class to be reliable, usable, and economically viable. Today, the Smart Dust class lies between these two milestones.



(a) Volumetric evolution of classes. (b) Approximate first instances of each class.

Figure 1.3: **Corollary to Bell's Law: Roughly every decade, general-purpose computing grows an order of magnitude smaller.** In each case, the new class of computing is foretold by one or a few instances made a decade earlier, which proves such machines are possible. This dissertation looks at what is needed to move the millimeter-scale computer class from possible to prevalent, with an eye towards the even smaller systems to come.

1.2 Smart Dust & COTS Motes

The first major push to develop Smart Dust ran roughly for a decade around the turn of the century [1]. That project resulted in the miniaturization of many key elements of future Dust systems including sensor components, micro-actuators, communication frontends, and microprocessors [21–25]. In parallel to the millimeter-scale push ran a side project looking into the development of so-called Macro Motes[1] [26]. While not the primary focus of the original Smart Dust project, these motes had perhaps the more significant immediate impact.

Macro motes laid out the basic components of a useful sensing system. Paraphrasing and slightly updating the original definition: a system must have enough **energy** to do something useful for a reasonable amount of time, must have **sensors** to bridge the digital and physical world, must have **computation** capacity to operate intelligently (or at all), and must have **communication** faculties able to share its results. By showing that such systems could be realized at the centimeter-scale, out of readily accessible commercial-off-the-shelf (COTS) components, the mote computing class was born.

[1] "Macro" Motes were likely still some of the smallest complete computational sensing systems, but given the aspirations of the Dust team, all things are relative. Today, we generally call such systems simply "motes."

With the realization of proto-Dust and the first instances of mote-class computing came the same fundamental call to arms that echoes through this document. Kahn, Katz, and Pister asked, now that we are capable of building the proof-of-possible for this emergent mote class of computing, what will be required to go from one to many and to make these many new mote-computers useful [27, 28]? Hollar coined the term "COTS dust" to describe his centimeter-scale Macro Motes [26]. Embedded in this name is the revelation that modularity would not be a challenge for this computing class—it was defined as systems that had all of the desired functionality of the imagined millimeter-scale systems, but could be realized immediately via the composition of available compute, sensing, communication, and power modules. Instead, the need was to "[get] more systems-level researchers interested in this critical area."

Today, systems-level research in mote-class computing, i.e. wireless sensor networking research, is flourishing. As of this writing, the top two subject areas in ACM SIGMOBILE are "Wireless access networks" and "Mobile networks," both of which look to solve the communication, access, and control challenges for portable computational systems. The *International Conference on Information Processing in Sensor Networks (IPSN)* will celebrate its twentieth anniversary next year, and the *Conference on Embedded Networked Sensor Systems (SenSys)* will celebrate its nineteenth year. Key challenges spanning time-synchronization [29], medium access control [30], low-power design [31], networking [32], and much more have been addressed and continue be refined. Yet, if we were to try to make millimeter-scale motes today, we still face the same limitation as the original effort—there is no library of millimeter-scale modules from which to compose such systems.

1.3 The Case for Modularity

The spectacular success and diversity of today's centimeter-scale motes, and let us include much of the "Internet of Things" in this measure, are the evidence of the power of enabling modular design. On the one hand, there is proof in various successes at the manufacture of monolithic systems over the years that modular design blocks are not *necessary* for the creation of a millimeter-scale system [33]. On the other hand, that fact that such designs talk in the number of years to make one system rather than the number of unique systems they can make per year (or month or even day) is the essence of the scale that is missing. Today, a casual hobbyist can visit a local makerspace and in one afternoon devise, design, synthesize, test, and deploy a new centimeter-scale device.

This capacity for system-design-time synthesis of complex motes is enabled by the modular part ecosystem. Consider a radio IC, which encapsulates complex modulation and demodulation operations, precise timing requirements, and sensitive high frequency and analog design blocks into a stable, re-usable package with a lightweight, high-level interface. A system integrator can fuse this radio with a microcontroller, camera, and battery on one device and a microcontroller, screen, and battery on another device to build something new and quite complex like a virtual window, without requiring rich understanding of the internal

workings of any of the constituent pieces. Modular design "increases the range of 'manageable' complexity" and "accommodates uncertainty" [34]. Systems do not need to be built from the ground up every time, and the exact function and operation of a system can be deferred until system-design-time.

Interconnects and Their Role in Modular Design

Once a system is broken into modular pieces, there must also be something that connects modules back together into a system. In many cases, this interconnect can come to define a class of systems. Indeed, the "PC revolution" and the emergence of the PC class at scale is often defined by the emergence of a wide array of *IBM PC compatible* clones. To quickly and economically bring a new machine to market (the IBM PC and later the IBM XT and IBM PC/AT), IBM designed machines that pieced together generally commercially available modules. To build a clone then, manufacturers simply had to replicate the interconnect used to compose commodity hardware pieces.[2] The AT interconnect would eventually evolve by fiat from a pseudo-standard to the sanctioned Industry Standard Architecture (ISA) Bus, which defined the composition of computers for decades.

Modularity in the System on Chip Era

Despite the success of ISA and its descendants in the design of innumerable computing systems, today we are seemingly seeing a push back towards integration with the rise of the System on Chip (SoC) architecture. The integration of multiple modules onto a single die for fabrication does not, however, mean that the system which happens to be manufactured into a single physical chip is not modular. SoC designs usually include some form of internal interconnect network or bus (i.e. ARM's AMBA) which integrate modular hardware blocks through a narrow memory bus interface. This memory bus is usually strictly internal to the SoC, though various peripherals with external I/O capability may hang off this bus. This design provides encapsulation and strong isolation of the internal modules, but it does so at the cost of flexibility and composability; it closes off this bus to expansion and integration beyond the originally envisioned SoC. In Chapter 4, this dissertation will show how we can extend the system bus metaphor to become a hierarchy of interconnected system buses between higher-level modules. This allows the preservation of both composability between fabricated modules and the performance advantage of tight, on-die integration.

[2] At the time, the real challenge in creating a clone was replicating the Basic Input/Output System (BIOS) firmware. The original intent for BIOS was to act as a runtime hardware abstraction layer, which meant that clones had to provide this same interface in order for existing software to run. Accessing hardware through BIOS proved slow and inefficient compared to directly accessing hardware, however. Through a combination of performance and compatibility concerns (what is the abstraction layer that sits above the differing BIOS abstraction layers...), this hardware abstraction moved up into operating systems and their drivers, relegating BIOS to just the earliest part of the boot process in more modern machines.

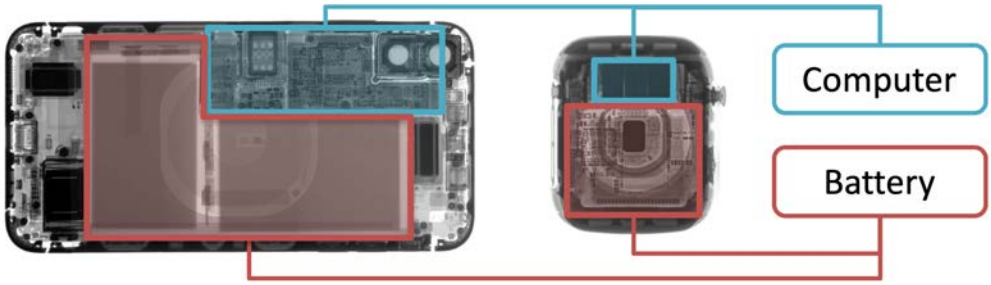


Figure derived from phone [36] and watch [37] images from iFixit. Figure licensed CC BY-NC-SA. Source available at patpannuto.com/dissertation.

Figure 1.4: **Looking inside modern computing devices.** Energy storage technologies are not shrinking at the same rate as computing. As a result, as computing devices get smaller, a larger proportion of their volume is dedicated to energy storage.

1.4 Energy Constrained Computing

In the halcyon days of early computing systems, the primary focus of system design was building bigger and better computers that could do more. With the emergence of portable computers (the suitcase-like precursors to laptops), computing 'cut the cord.' Suddenly, it became possible and reasonable for a computer to no longer be tethered to a wall, and equally suddenly it became important for these machines to be able to do more with less. The power draw of a system now determined the lifetime of a system (at this point in history, power dissipation and resulting thermal management concerns were also on the radar, but not yet a primary limiting factor for most semiconductor system performance [35]).

Again, there is a corollary available to Bell's Law. Instead of considering a constant price in dollars and cents, one can hold constant the energy reserves of a portable machine, and find that time and its technological improvements allow for more capable machines (i.e. the steadily improving laptop class). Similarly, one can lower energy rather than lowering price, and find devices slowly shrinking for about the same capability and the possibility of sub-class formation (i.e. thinner, lighter laptops and lately the blurry line between these and tablet computers). Finally, one can lower energy dramatically, and find a new class of device, with comparatively less compute capability, but new levels of portability and utility (i.e. the emergence of the smartphone class).

Interestingly, this trend is driven both by the reduction in energy demand of newer compute technologies as well as the increase in capability of energy storage technologies. Indeed, battery technology has been improving for centuries, since Alessandro Volta's first voltaic pile in 1799. One consequence of this long history, however, is that today the rate of improvement for energy storage technology is considerably slower than that of computing [38]. As a result of the miniaturization of computing outpacing the miniaturization of energy storage, it is increasingly becoming the case that the principle dimension of a computing system is dictated by the energy required to power it, as seen in Figure 1.4.

Techniques to Reduce Energy

Smart Dust pushes the limits of today's capabilities to miniaturize computing. As a result, these systems no longer adhere to the volume proportionality of current portable computing systems. The systems in Figure 1.1 are more computer than battery. To operate then, these systems must use proportionally less energy as well. That is, Smart Dust systems must develop new energy reduction techniques beyond those used in portable computing today.

Voltage and Frequency Scaling

When a system is active, its power use is dominated by switching power losses. Switching power draw is proportional to the square of the system voltage and to the system frequency (i.e. $P = C \times V^2 \times f$). This relationship drove significant effort into reducing system operating voltage to, and later below, the threshold voltage of standard CMOS [39]. There is a bound to the performance gains of voltage scaling, however. As voltage decreases, leakage current grows, and eventually the gains in switching performance are overcome by this increased leakage [40]. The chips used to compose the Smart Dust systems, as described in this work, reach a core voltage of 0.6 V, which is near the limit of today's practical fabrication technologies. While there is still some headroom, particularly with the advent of lower-threshold low-leakage transistor technologies, voltage scaling alone is insufficient to reach energy targets.

Somewhat counter-intuitively, low-power systems often aim to increase their operational frequency (within reason). While increasing frequency increases the instantaneous power draw of the system, it also allows the system to complete its operation in a shorter window of time. Ultimately, the goal is to reduce the total energy use—which is the integral of power over time. The insight here is that in a vacuum, because power is linearly related to frequency, changing the frequency up or down would have no net change in consumed energy. However, in real systems there are additional, constant, static overheads. This is the principle that underlies computational sprinting [41]. Of course, generating very fast clocks is also energy intensive, which caps the practical benefits of increasing frequency as well. The takeaway now is that while voltage and frequency scaling both can help, they alone are insufficient to realize Smart Dust energy budgets.

From Dark Silicon to Pitch Black Silicon

What happens when a system finishes its computational sprint? Systems will shut down components that are not in active use to save energy. In performance-oriented designs, this phenomenon where more and more die area is shut down is commonly referred to as "Dark Silicon" [42]. For these systems, shutting down generally means cutting the clock that feeds various subsystems or asking components to enter a sleep state to eliminate idle switching losses. However, these 'shut down' component still preserve state and can exhibit static leakage that proves non-trivial for the energy budget of Smart Dust systems. This is where the extremely resource constrained Smart Dust class exhibits a new operational paradigm, which I call **pitch black silicon**.

When subsystems are not in use, Smart Dust systems power them off completely. Without a new support mechanism, this aggressive energy management strategy can sit at odds with modularity. Components and subsystems must now be freestanding and can no longer rely on the presence of an always-on, central power management block. For the circuit designer, this means each module must provide its own custom timing and control signals to manage cold boot; it also means a new, potentially unique-per-chip, interface to report and control its power state—how, when, and why does something power on? For the systems designer, this turns power state into a distributed systems problem—how is one module to know whether it can safely communicate with another or if it must first power on its target? For the platform integrator, a series of ad-hoc answers to these questions limits the flexibility and reusability of components—routing custom power control between modules is no longer modular design—and impedes abstract, platform-level functionality—how does something like a brown-out detector quickly reduce system load? Modular design with pitch black silicon requires that energy and power are managed by the interface that defines module boundaries.

1.5 Thesis Statement

With the introduction of a clockless, "shoot-through" ring topology, it is possible to design an interconnect that both encompasses the modular design principles that enable system-design-time composition and respects the resource constraints of millimeter-scale "Smart Dust" systems; the creation of such an interconnect will accelerate the development of a diversity of Smart Dust systems and accelerate the actualization of the millimeter-scale computing class.

1.6 Contributions of this Dissertation

This dissertation presents modular design principles, and the development of a new interconnect to support them, for the Smart Dust computing class.

The foundation of this dissertation is MBus. MBus is part of the larger Michigan Micro Mote (M3) initiative, which aims to build an array of novel micro-scale motes—Smart Dust—and works to develop all of the constituent technologies required to realize these devices. Within this larger whole, the development of MBus was a collaboration with Yoonmyung Lee, Ye-Sheng Kuo, ZhiYoong Foo, Benjamin Kempke, Ronald Dreslinski, David Blaauw, and Prabal Dutta. Various pieces of MBus described in this document have been published at CICC'14 [43], ISCA'15 [44], and Micro Top Picks '16 [45].[3] In addition, some of the initial exploration that drove the need for MBus was published as a demo at IPSN'12 [50]. Ideas and results from these publications are woven through the entirety of this document, primarily Chapter 2, Chapter 3, and Chapter 5 and to a lesser extent Chapter 4. In contrast to the prior refereed proceedings, this dissertation is able to take a holistic view of the entire design.

[3]There are several additional publications related to MBus, and moreso the systems built on top of it, whose content are not included in this dissertation (non-exhaustively [46–49]).

Chapter 2 lays out the case for a new bus. It looks at an expanded history and comparison of embedded interconnects, including the addition of newer designs such as I3C (MIPI), to understand the root causes of overheads and incompatibilities for Smart Dust. It then synthesizes the needs from circuits to systems to lay out the set of criteria for an interconnect for resource constrained systems. These criteria then serve as the foundation for the rest of the design and evaluation.

Chapter 3 describes the design of MBus. The design is broken into three major elements. The first design point is the physical interconnect, in which a new "shoot-through" ring topology is presented as the best balance of I/O overhead, system active power, and scalability. The second design point is the logical operation of the interconnect, which looks at how to safely and efficiently share the physical wires and also introduces a new, low-overhead interjection technique to handle message framing, acknowledgment, and bus errors robustly and with very low overhead. The final design point is new to interconnects and a unique requirement of Smart Dust systems. Here, MBus adds faculties for transparent, complete system power management. The interconnect is able to provide the illusion of always-on components, while eliminating nearly all dynamic and static leakage power losses by completing powering off unused components and managing their cold-boot in parallel to message arrival.

Chapter 4 describes the design of MPQ, the protocol that runs atop MBus and defines the interface between modules. Before laying out the protocol design, this chapter looks at the tradeoffs between standardization and capability afforded from interconnects across a broader landscape of computing than previously discussed. With this background, it makes the case for a standardized model of memory and registers across all connected devices, and further a standard means of controlling data transfers. The result is a novel and powerful namespaced MMIO abstraction. This interaction model will ultimately serve as the foundation for the federated operation highlighted in Chapter 5 as well as the system-level programming and debugging techniques laid out in Chapter 6.

Chapter 5 makes it all work on real hardware. It begins by mapping the concepts from Chapter 3 and Chapter 4 onto concrete hardware designs. This section covers some of the key subtleties and implementation details. Then it evaluates the performance of the interconnect, both in principle and in practice. This includes theoretical analysis of the performance of the protocol and measured performance on two real-world Smart Dust systems: a temperature sensor and an imager.

Finally, Chapter 6 zooms out from individual devices and looks at the Smart Dust ecosystem as a whole. It describes the design of M-ulator, a new hybrid simulation and in-circuit emulation tool designed to aid the initial bringup and debugging of Smart Dust systems. It also looks at questions of programming and debugging Smart Dust once physical access is no longer an option. Here, the platform design from Chapter 4 shines, as a minimalist injection frontend is able to support rich, complex operations such as *in situ* programming and restricted debugging.

Chapter 2

The Case for a New Bus

This chapter begins with an in-depth look at existing interconnects. The goal is to understand common design points and what aspects do and do not work well for the Smart Dust computing class. Ultimately, this survey concludes that no existing design can meet all of the needs of Smart Dust. This summary is then combined with experience from building a number of pre-modular Smart Dust systems to draft a list of requirements for the interconnect for this emerging computing class. This chapter makes no design decisions, rather it seeks to lay out the constraint space that will inform the architecture of modular, millimeter-scale computing.

2.1 Related Work

Before building a new bus, it is important to first show why existing, widely adopted bus protocols like I^2C, CAN, SPI, and I^2S are not viable system interconnects for modular, resource-constrained systems. The problems stem from their energy use, protocol overhead, pin count, and system design requirements.

I^2C, 1-Wire, CAN, and Other Open-Collector Buses

Many interconnects use an open-collector or open-drain design (e.g. I^2C [51], SMBUS [52], and CAN [53]). This circuit construct turns each bus line into a wired-AND; one or many devices can drive a `0` on the bus, but if nothing actively drives low, then pull-up resistors pull each line high. The advantages of this approach are decentralized arbitration and multi-tiered priority. The pull-up resistors, however, are not energy efficient and result in designs that have up to three orders of magnitude worse energy per bit than MBus.

To illustrate, consider an idealized I^2C configuration running at 1.2 V (a typical I/O voltage for Smart Dust) that tries to optimize for energy consumption. I^2C typically requires the pull-up resistor be sized to accommodate 400 pF of total bus capacitance, but that can be relaxed to, say, 50 pF for micro-scale systems; fast mode I^2C has a 400 kHz clock and must reach 80% V_{DD} in 300 ns, but again relax that (eliminate setup and hold time) to the full

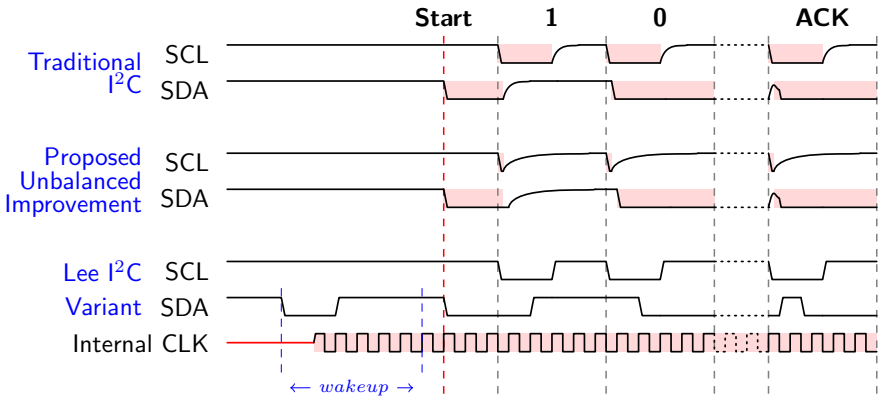


Figure 2.1: **Waveforms of I²C and variants.** A comparison of traditional I²C and some of the proposed variations. Shaded areas are power-expensive protocol elements.

half-cycle (1.25 µs). This relaxed I²C bus requires a pull-up resistor no greater than 15.5 kΩ. To generate the bus clock, this resistor is shorted to ground for a half period, dumping the charge in the bus wires, pads, and FET gates (23 pJ) and dissipating power in the resistor (116 pJ). The clock line then floats for a half cycle and the resistor pulls it high (35 pJ). Thus, generating the clock alone draws 69.6 µW. Eliminating the switching power—the 23 pJ/bit charging and discharging of the wire, pad, and gate capacitance—requires complex adiabatic clocks, outside the scope of many practical designs [54, 55]. MBus finds its energy gains by eliminating the 151 pJ/bit lost to the pull-up resistor.

I²C Variations

One conceivable idea for reducing the impact of the pull-up might be to "unbalance" the clock as shown in Figure 2.1. This would allow the designer to nearly double the size of the pull-up resistor (halving the power draw) while maintaining the same bus clock period and minimizing the impact of the SCL line on energy usage. Unfortunately, this concept does not reduce the energy consumed by the pull-up while pulling up, nor does it reduce the energy consumed by the data line when transmitting `0`'s. Unbalanced clocks would also require local timing modifications, costly in energy and complexity, ruling out this possibility.

Lee et al. reduce I²C power draw by designing an "I²C-like" bus that replaces the pull-up resistor with logic that actively pulls the bus high and a low-energy "bus keeper" circuit that preserves the last value [17] (similar to I²C Ultra Fast-mode [51]). While this eliminates the pull-up, it does so at the cost of requiring a local clock running 5× faster than the bus clock, the energy inefficiency seen in Figure 2.1. There is not a clear path to designing an I²C or I²C-like bus without either a pull-up or a fast-running internal clock. While the internal clock is not as energy-intensive—Lee's system is able to reduce bus energy to 88 pJ/bit (4 times that of MBus)—, Lee's design also requires hand-tuned, process-specific ratioed logic; this requires manual tuning of every chip and runs counter to the goals of a general-purpose bus.

Furthermore, while Lee's I^{2}C variant is designed with commercial interoperability in mind, in practice actual interoperation requires an FPGA to translate between I^{2}C and the "I^{2}C-like" bus [50]. MBus eschews the "partial compatibility" that I^{2}C-like buses provide and uses the clean break to reconsider the primitives provided by the system interface, allowing the addition of features such as power oblivious communication, broadcast messages, and efficient transaction-level acknowledgments.

SPI, I^{2}S, Microwire, and Other Single-Ended Buses

As single-ended buses, SPI and its derivatives do not suffer from the power challenges faced by open-collectors and have little to no protocol overhead. SPI, however, requires a unique chip-select line for every slave device. In a modular system with a variable (and unknown until *system design* time) number of components, it is difficult to choose the "right" number of chip select lines a priori—too few impede modularity and too many violate the area constraints of Smart Dust systems. Additionally, SPI requires a single master that coordinates and controls access to all slave devices, and it requires all communication between slave devices to go through the master node. This more than doubles the communication cost for slave-to-slave transmissions: every message is sent twice plus the energy of running the central controller. A further subtle, yet critical, implication of a single-master design is that all communication is master-initiated. For a sensor to signal a microcontroller (i.e. an interrupt), it requires an additional I/O line, a resource that is unavailable to Smart Dust systems.

Alternative configurations such as daisy-chained SPI can eliminate the chip-select overhead but do not solve the multi-master/interrupt issue and require the addition of a protocol layer to establish message validity. As a system-wide shift register, a daisy-chain configuration adds overhead proportional to both the number of devices and the size of the buffer in each device. SPI and its derivatives are fundamentally incompatible with size-constrained microsystems.

The Improved Inter Integrated Circuit (I3C)

Some of the limitations of I^{2}C and SPI are also affecting the emerging mobile (smartphone) and wearable computing classes. The MIPI Alliance is a trade association that aims "to design and promote hardware and software interfaces that simplify the integration of components built into a device [56]." In 2014, recognizing the limitations of I^{2}C and SPI, MIPI formed a working group to develop a new interconnect technology. In 2018, the result was the Improved Inter Integrated Circuit (I3C) [57].

The I3C design has many similarities to Lee I^{2}C. In particular, it also introduces the idea of both a bus keeper and explicit push-pull logic. However, I3C includes extra provisions that attempt to better ensure backwards compatibility with I^{2}C. The I3C insight is to assign a fairly capable "Main Master" device that has full knowledge of all attached devices as well as the ability to dynamically detach the pull-ups. At idle and in early arbitration, the bus always uses the pull-up resistors in the traditional I^{2}C manner. However, during addressing if the Main Master establishes that no legacy I^{2}C devices are involved, it can detach the

pull-ups and switch to active drivers. While this (and some additional details in the I3C specification) does resolve the interoperability challenges, it still requires I3C member devices to include local high-speed clocks, just like Lee I^2C. Indeed, the I3C specification suggests that it achieves an approximately 10× reduction in energy per bit over traditional I^2C, with a best case around 100 pJ/bit, which is very similar to Lee I^2C's 88 pJ/bit. If, however, a bus were to eliminate the need for local oscillators on each node, the active power can be reduced another full order of magnitude, a necessary energy savings for Smart Dust class devices.

Another interesting aspect of I3C is the partial support for power-conscious nodes and systems. The specification includes some considerations for behavior when system or node power states change, however "system power management" is explicitly listed as not in scope for I3C. Instead, the bus adds support for "Hot-Join", where nodes may dynamically leave or join the bus (including the master, which must delegate another master before powering off or possibly leave the bus inoperable until it powers back on). While this design allows nodes to power themselves off and on, it does not provide any in-band mechanism to remotely power on another node, which means that systems must provide additional I/O signaling if it is expected that one node be able to request that another node power on. In the extreme, full any-to-any power requests would require a complete parallel interconnect network solely for system power management. To support full flexibility in power management policies, it is necessary for system power management to be supported in something that connects all nodes, which means either that the system interconnect must support power management or its connectivity must be duplicated.

The other major change from I^2C is the addition of dynamic addressing. As I3C retains the 7-bit address space from I^2C, it resolves the potential address conflict problem by requiring that nodes use dynamic address assignment. While this is useful for systems where conflicts occur, the requirement to always use dynamic addressing introduces needless overhead in systems where pre-assigned addresses would not have conflicted. This overhead is particularly troublesome in conjunction with the Hot Join mechanism, as the specification requires that dynamic address assignment occur at every start-up event (and that full reassignment occur whenever the master node starts up).[1] In Smart Dust systems, whole nodes power off and on frequently to realize energy goals, which makes such a re-enumeration requirement poorly suited to these designs.

Bus Designs from Other Disciplines

The original token ring protocol requires passing empty frames so that nodes can grab the token when they need it. Low-power systems rely on low duty cycles to remain efficient. Using tokens in place of arbitration either requires occasional empty frames to pass the token, with an inherent latency/energy tradeoff, or a sacrifice of in-band multi-master capability.

[1] One positive aspect of this design is that it partially ameliorates one system power management challenge. By snooping, other nodes on the bus could know when a powered off node powers on. This is of limited use, however, as other nodes still have no way to request that a node powers on, simply the ability to observe that it has happened.

The Domestic Digital Bus (D2B) [58] is a ring topology, like MBus. D2B links are high-power fiber optics, targeting automobiles. D2B requires a separate electrical network to facilitate wakeup of the optical frontends and uses a centralized master that must know the complete topology in advance [59]. D2B is not easily adaptable to the micro-scale domain.

Some of the new network-on-chip (NoC) protocols that have been developed, such as Nehalem's QuickPath [60], include power-aware features like MBus. These buses seek to move large amounts of data often via wide parallel buses, within a siloed system rather than compose independently designed, modular components.

Recently, work in energy-efficient, short-reach data links has led to energy performance as low as 0.54 pJ/bit [61]. These designs target high-performance computing applications, however, utilizing complex transmit and receive circuitry with high-speed clocks [62] and add requirements such as a common substrate with carefully carved channels [61]. This limitation is not well suited to a platform bus, which aims to support a wider diversity of packages and physical interconnection technologies.

Power Savings Create Communication Problems

Smart Dust systems need to aggressively conserve power. Devices that are left on or in standby allow communication to occur on-demand. Ultra-low power systems, however, realize their power goals in part through aggressive duty-cycling. A power-gated node must be awakened before it can communicate, presenting interoperability (how to wake a node) and run-time (when to wake a node) challenges.

Lee et al. identify this wakeup issue and modify their protocol to include a wakeup signal: an I^2C start bit followed shortly by a stop bit. This requires the sender to know the receiver's power state in advance or to unconditionally send the wakeup sequence before every message. Due to implementation choices, the minimum time between the start and stop bits of the wakeup sequence and the time until the chip is awake after the stop bit is received varies from chip to chip, which requires hand-tuning and conservative estimates. This design also requires each chip to implement a power-on circuit capable of self-starting.

In contrast, MBus takes over the power management of nodes, freeing designers from the burden of building complex self-start circuits. MBus guarantees delivery of messages, independent of the power state when a message is sent, eliminating requirements for distributed power state management.

The Opportunities of Clean Slate Design

One lesson from the Lee I^2C and I3C designs is that working from an existing standard can place limitations on design that are not required by the underlying system. For that reason, the next section takes a constructivist approach. It looks from the perspective of both a circuit designer and system designer at what properties an interconnect might provide to support Smart Dust systems. In the end, mapping those requirements back to these buses will find that no current design meets all the needs of Smart Dust.

	I²C	SPI	UART	Lee-I²C	I3C	MBus
Driven by Physical Limits						
Synthesizable	**Yes**	**Yes**	**Yes**	No	**Yes**	**Yes**
I/O Pads (n nodes)	**2/4**†	$3 + n$	$2 \times n$	**2/4**†	**2/4**†	**4**
Global Unique Addresses	128	—	—	128	—	2^{24}
Multi-Master (Interrupt)	**Yes**	No	No	**Yes**	**Yes**	**Yes**
Standby Power	**Low**	**Low**	**Low**	**Low**	**Low**	**Low**
Active Power	High	**Low**	**Low**	Moderate	Moderate	**Low**
Driven by Protocol						
Data-Independent	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Robust Bus Reset	No	Yes	No	No	No	**Yes**
Dynamic Address Option	No	—	—	No	**Yes**	**Yes**
Hardware ACKs	**Yes**	No	No	**Yes**	**Yes**	**Yes**
Broadcast Messages	Option	Option	No	No	**Yes**	**Yes**
Power Aware	No	No	No	No	No	**Yes**
Bits Overhead (n bytes)	$10 + n$	**2**‡	(2-3)§ $\times n$	$10 + n$	$10 + n$	**19**, 43*

† When wirebonding, a shared bus requires two pads/chip (or a much larger shared pad)
‡ Asserting and de-asserting the chip-select line
§ Depending on the stop condition; assumes 8-bit frames and no parity
* Depends on whether short (more common) or long addressing is in use

Table 2.1: **Feature comparison matrix.** Population-independent area, ultra-low power operation, synthesizability, an area-free global namespace, and interrupt support are fundamental requirements for a general purpose Smart Dust interconnect. Standby power is on the order of 100's of pW and active power ranges from 10's of µW to 10's of nW.

2.2 Interconnect Criteria

This section summarizes the requirements for the system interconnect of ultra-constrained systems. These requirements and guidelines are the product of several generations of experience in building such devices.[2] The requirements are divided into two major classifications. The first set describe requirements driven from physical-world realities that impose inviolable limits on the design space. The second set describe the needs of systems and inform direction within the allowed physical design space. Table 2.1 summarizes these requirements.

[2]I wish to emphasize again here that this is a summation of a collective experience. I am extraordinarily grateful to all members of the Michigan Micro Mote team, especially David Blaauw, Prabal Dutta, ZhiYoong Foo, Benjamin Kempke, Ye-Sheng Kuo, and Yoonmyung Lee whose ideas and insights I sincerely hope to have represented well here. Additional thanks to the members of the Berkeley Wireless Research Center, the TerraSwarm Research Center, and the Stanford Internet of Things Project, who provided critical early and late-stage feedback.

Physical Constraints

Smart Dust is largely defined by being physically small. This means that there is limited surface area for the individual modules that make up a system and limited volume for the whole of a system. As a result, systems have limited energy and are restricted in their physical design. This section distills these physical realities into constraints on interconnect design.

Low, Fixed Wire Count

With sub-millimeter scale systems, the area cost required to place bonding pads (35–65 µm wide) on one edge or around the perimeter of a chip limits the ability of the system to scale. While advancements such as through-silicon vias (TSVs) help, many popular processes do not support them (e.g. IBM130 or TSMC65). In conventional designs, ICs aiming to maximize flexibility can simply over-provision GPIO pins to allow for potentially large numbers of peripherals. Even at just 50 µm^2 each, four I/O pins[3] are already 1% of the total area on a 1 mm^2 chip, however. Indeed, packaging techniques are not minimizing as quickly as other aspects of design, and the result is an increase on I/O pressure as systems shrink. Due to this I/O pressure, over provisioning GPIOs is often not a reasonable design choice.

A fixed wire count maximizes adaptivity. If the physical topology of a bus were to vary based on the number of attached nodes, then the maximum number of devices in a system is a decision made at chip fabrication-time. Instead, if the wire count is fixed and independent of the number of attached nodes, then composition can be a design-time consideration. This relieves the pressure on chip designers to accurately predict the "correct" amount of extensibility that will be needed and frees system designers to use chips in potentially novel and unexpected ways.

Address Space

A bus must provide a way of addressing each node in the system. One of the simplest approaches is to include dedicated hardware for this purpose, such as the chip-select mechanism in SPI. Such a technique, however, requires sufficient I/O availability to make a unique connection to each peripheral. Dedicated select pins place a hard limit at fabrication-time on the number of directly addressable[4] nodes. As stated, this is an incompatible property for a general-purpose bus in I/O-constrained ecosystems.

If hardware-based node selection is not available, then the protocol must include some addressing scheme. The length of the address presents a tradeoff between the overhead of addressing and the total number of uniquely addressable nodes. While there are likely to be only a modest number of nodes in any given system, addresses are fixed at fabrication-time,

[3]Note here that buses with an electrically connected, shared line (e.g. I^{2}C's SDA and SCL lines) require two bonds per chip on all but the ends of the system as two wire bonds cannot land in the same place. In the context of wirebonding, I^{2}C is better thought of as a 4-wire bus rather than a 2-wire bus.

[4]While techniques such as daisy-chaining can allow for greater flexibility, they do so at significant runtime cost, and require additional support from all involved peripherals.

and thus must be globally unique to maximize composability. Certain I^{2}C devices afford limited design-time flexibility by exposing one or more I/O pins that designers can tie to set a few address bits and thus resolve potential conflicts. An interconnect for severely I/O-limited systems cannot rely on this mechanism, however, and must size its address space accordingly.

Interrupts / Multi-Master

To facilitate a diverse and unpredictable set of devices and system applications, any device must be able to initiate a transmission to any other device at any time. With a fixed number of wires, dedicated signaling to a central controller (such as the interrupt lines in SPI) is not viable. This means the protocol requires either an efficient, non-polling based interrupt mechanism or a true multi-master design.

Low Standby Power

Resource-constrained systems spend the majority of their time in standby so standby inefficiencies are magnified. Existing interconnects are well suited to this, so any new bus must draw less than 100 pW to be competitive.

Low Active Power

Smart dust systems have extremely constrained power budgets. While any absolute power number will be system-dependent,[5] to allow Amdahl-balanced system design, the interconnect cannot draw significantly more than other components. Realizing this active power constraint informs additional restrictions.

No Local Time Creating a stable, local time base (that is, powering an oscillator) is energy-expensive. If a node, such as an analog circuit acting as an event detector, would otherwise not require an oscillator, adding one simply to support the interconnect can quickly dominate the energy use of that chip. In an analysis of the Lee I^{2}C variant, the majority of energy used by the interconnect goes to powering local oscillators on each node. While there must be a time base somewhere, to supply the bus clock, to realize low active power in the aggregate it is important to not require a sense of time on each individual node.

Minimize Idle Listening Following state can also be energy expensive. Again a lesson from Lee I^{2}C, while a bus is necessarily shared among many nodes, point-to-point communication is the most common use case. Nodes should be able to quickly establish whether they are involved in a transmission (addressing should be first) and should have minimal

[5] As a point of reference, one of our most constrained systems is powered by a 0.5 µAh battery and targets a total system active power budget of < 40 µW (and idle power of 20 nW). In practice, the MBus design targets a hard upper limit of 20 µW total active power draw for the system interconnect. Generating a traditional I^{2}C clock signal alone for the given system would require approximately 70 µW.

overhead in transactions they are not involved in. This principle will guide protocol design, as non-participant nodes should not need to track every byte of long transactions to establish when the bus is again idle and available for use.

Synthesizable

To facilitate widespread adoption, designs must be process-agnostic. Concretely, it must be possible to describe them as a block of "pure" HDL with no process-specific custom macros. While this is a common property of established interconnects, per-chip overhead is much less costly in early research prototypes. Indeed, the Lee I^2C variant included delay chains that required characterization to satisfy timing requirements. At scale, process-specific tuning of custom ratioed logic adds cost, complexity, and risk to every implementation.

Additional Considerations

The physical constraints limit the physical design space for the interconnect. However, there are still many degrees of freedom within the low-I/O, low-power design space, particularly at the protocol level. As systems are composed and richer and more complex interactions are required, higher-level primitives become more important. This section describes properties that are critical to supporting composable, usable, reliable systems.

Data-Independent Behavior

In order to accurately model and predict bus performance, it is important that its operation be deterministic. Many protocols use dedicated symbols to communicate special cases (such as an end-of-message indicator). Supporting such sentinels requires byte stuffing, which in pathological cases can double the length of a message. This affects the ability to reason about protocol performance both in energy and time, and in real-time systems can lead to violations of timing requirements or require artificially high provisioning. Instead, the behavior of a protocol must not be affected by the data it is transmitting.

Fault Tolerance

Transient faults are inevitable. While robustness against short-term corruption is desirable, it is critical that recovery from such corruption is possible and it is impossible for the bus to enter a "locked-up" state due to any transient faults. Furthermore, the recovery mechanism must respect the previous constraints. The area (wire count) constraint requires that fault recovery be in some way in-band. However, the data independence constraint disallows the use of any form of sentinel sequence. Finally, the low active power constraint suggests avoiding solutions that rely on a local sense of time, which eliminates traditional timeout-based approaches. In practice, robustly and efficiently supporting fault tolerance proves one of the most difficult aspects of bus design.

Efficient Acknowledgments

Many applications require reliable message transport. This feature may be directly supported by the bus protocol in hardware, or as an optional software feature if it can be made sufficiently low-overhead.

Broadcast Messaging

In principle, any shared communication medium could implement broadcast messaging. In practice, few interconnects exploit this capability. Part of the challenge perhaps is that broadcast support is historically an optional feature to which few designs opt-in, which reduces the utility of broadcast messaging, which further reduces incentives for future devices to opt-in; this cycle effectively renders the option moot. The overhead of supporting broadcasts is exceptionally low, however (simply a second address to match), and the potential benefit in message reduction is very high. Mandating broadcast support further enables the interconnect to specify self-management directives, which allows for runtime optimizations.

Power-Aware

Unlike deep sleep, which still loses energy to static leakage, a power-gated circuit loses all state. Ultra-constrained systems need to cold boot and later shut down sub-circuits without affecting active areas. Interfaces between power domains must be *isolated*, tied to a fixed value by an always-on logic gate, so that floating signals do not confuse active logic. To power on a power-gated circuit reliably and without introducing glitches, four successive edges, the *wakeup sequence*, must be produced:

1. Release Power Gate: Supply power to the circuit that is being activated.
2. Release Clock (If digital logic present): After a clock generator is powered on, it requires time to stabilize before driving logic.
3. Release Isolation: Outputs of a power-gated block float when off and must be isolated until they are stable.
4. Release Reset: Once stable, the circuit may leave the reset state and begin interacting with the rest of the system.

This sequence is fundamental to powering on sub-circuits.

Aggressively low power designs have no clock sources in their lowest-power state and no means to generate these signals. In current systems, every design requires a custom "wakeup" circuit to generate these edges, adding cost and complexity. These low-level details should be hidden from the application developer. The wakeup sequence provides a clean interface to power on a system and the system interconnect should provide a clean abstraction for sending messages, i.e. one that ensures receipt independent of the target device type or immediate power state.

Interoperability

Not all systems may be severely resource or energy constrained, yet they may still wish to use chips designed for ultra-low power applications. Any interconnect must bridge the gap between power-conscious and power-oblivious—no notion of power-gating and no specialized constructs to support it—devices to avoid fracturing the component ecosystem and to enable reuse across all device classes.

2.3 Summary

This chapter has identified why existing embedded interconnects are not suited to support modular design for the millimeter-scale computing class. It then takes a holistic look at the Smart Dust design space, from physical manufacture through integrarted systems operation, to lay out the criteria and necessary capabilities of an interconnect that could enable composable design for this class of systems. The next chapter will take these requirements as a foundation to drive the design a new interconnect, built for the millimeter-scale computing class.

Chapter 3

MBus

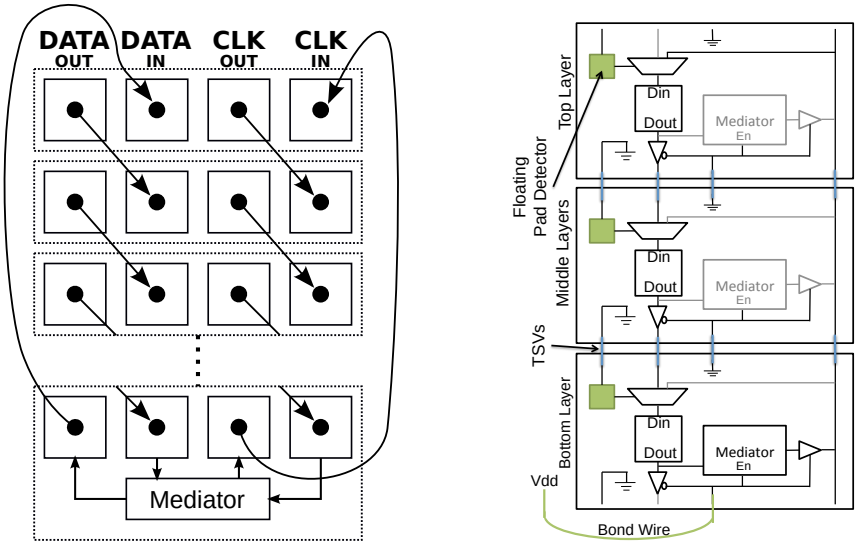
This chapter describes MBus, the interconnect that enables modular Smart Dust. This dissertation does not aim to be full specification for MBus. The MBus Specification and a reference implementation are available at mbus.io. Instead, this chapter explains the design decisions behind MBus, working from the constraints identified in the previous chapter.

3.1 MBus Physical Design

MBus is an interconnect that enables design-time synthesis of multiple physical chips and is optimized for minimal area and energy overhead. Every endpoint attached to MBus is referred to as a *node*. In the common case, every chip in a system corresponds to an MBus node. In principle, there is no limit to the number of nodes. The physical bus is made up of two signals, a bus clock (`CLK`) and a data line (`DATA`). These wires are connected in a ring and carry synchronous data.

MBus Topology: Efficient Rings to Share Wires

To meet the wire count requirement, the bus topology must be independent of the number of nodes, i.e. adding another node cannot require adding a new wire. As many nodes thus share the same wires, MBus requires a scheme to avoid conflicts, where different nodes drive the same line high and low. Some form of token-passing or leader-based protocol would violate the efficient interrupt requirement, as it would require the leader to poll to find the interrupter. MBus prevents conflicts by using a ring topology, as shown in Figure 3.1. There are two signals and thus two rings in MBus: `CLK` and `DATA`. In each ring, every connection is single-ended, which eliminates the possibility for conflicts. To minimize active power, MBus clocks all bus logic off of the bus clock itself. This obviates the need for a local oscillator on each node. With no local clock, the rings are "shoot-through": signals pass through only a minimal amount of combinational logic from one node to the next, as shown in Figure 3.2.



(a) 3D Stack Example (b) TSV Example†

†Only `DATA` shown for clarity, the `CLK` line uses identical selection circuitry.

Figure 3.1: **MBus topology.** An MBus system consists of a mediator node and one or more member nodes connected in two "shoot-through" rings. The ring topology adapts to many system synthesis methods, such as stepped 3D-stacking with wirebonding (a) and TSVs (b).

Circuit Techniques to Minimize Active Power

Regular MBus node are clock-less. Flip-flops in MBus nodes are purely triggered by $\texttt{CLK}_{\texttt{IN}}$. Hence, if the bus is idle, regular nodes draw only static leakage power. This is key since the addition of a simple clock generator to each regular node will quickly dominate total power consumption. Initially, only a small address detector is clocked by $\texttt{CLK}_{\texttt{IN}}$ which observes the $\texttt{DATA}_{\texttt{IN}}$ signal to determine if the node is being addressed. If a node is not the addressee, it directly forwards `CLK` and `DATA`. Any internal flip-flops used to store or process incoming data do not toggle. Simulations show that this design point reduces energy consumption by 23%.

Clock Generation and Bus Mediation

MBus introduces one special node, the *mediator*. The mediator is responsible for generating the bus clock and resolving arbitration. Every MBus system must have exactly one mediator, either attached to a core device (e.g. a microcontroller device) or as a standalone component (similar to the pull-up resistors in I^2C). For ultra-low power designs, MBus power-gates all but the forwarding drivers (Wire Controller) and a minimalist wakeup frontend (Sleep Controller). The mediator must therefore be capable of self-starting. In an ultra-low power design, something must have the capability to self-start; the mediator allows that self-start requirement to be contained within a single, reusable component. For the other nodes, bus operations are carefully designed to ensure that clock-less designs can fully participate.

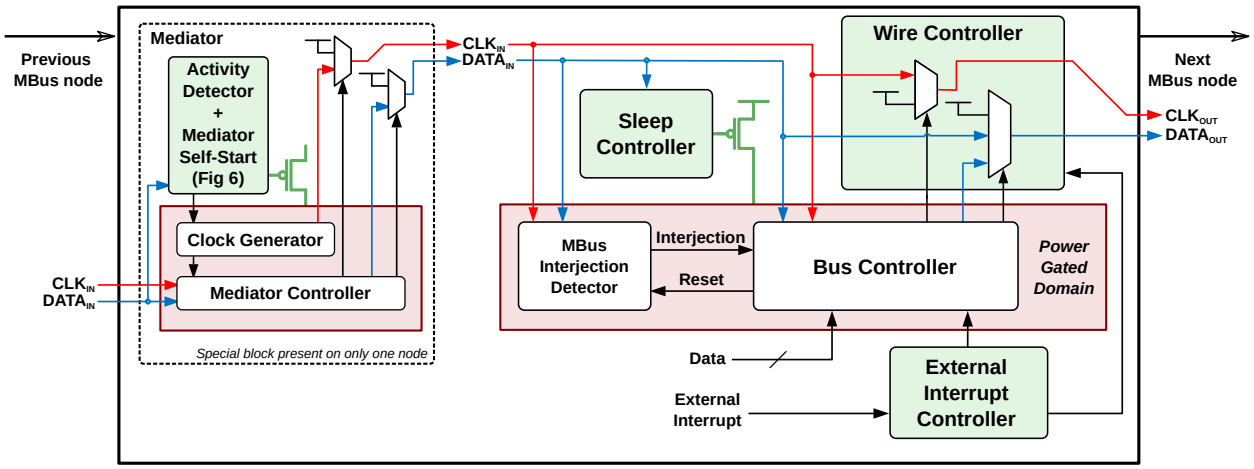


Figure 3.2: **Conceptual frontend.** A simplified view of the MBus frontend. The default configuration "shoots-through" DATA and CLK signals. Note that there are no local clocks or buffers (beyond generic I/O buffers and drivers) in the MBus ring.

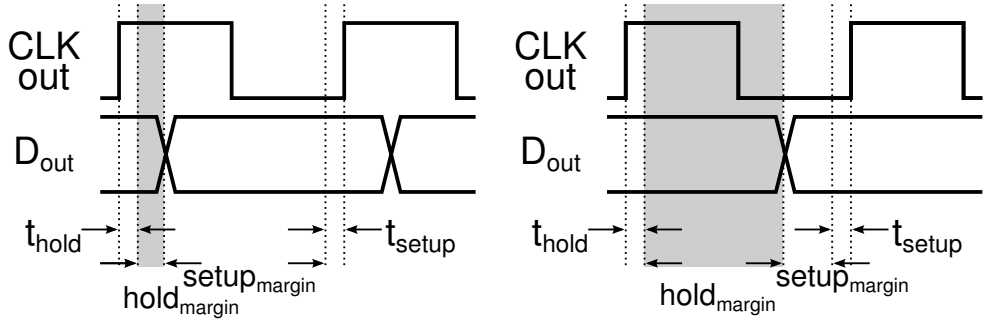


Figure 3.3: **Setup and hold time diagram.** Conventional positive-edge triggering is shown on the left while the balanced MBus clocking is shown on the right.

Robust Timing

In a modular system, the loading and driving strength of different $\texttt{CLK}_{\texttt{OUT}}$ and $\texttt{DATA}_{\texttt{OUT}}$ drivers is unpredictable, which creates uncertainty in the relative arrival time of $\texttt{CLK}_{\texttt{IN}}$ and $\texttt{DATA}_{\texttt{IN}}$. This requires the insertion of a large number of hold-time buffers, which would increase power draw and lower performance. Instead, MBus separates driving and latching edges to balance setup and hold time margins, as shown in Figure 3.3. $\texttt{DATA}_{\texttt{IN}}$ is sampled on positive $\texttt{CLK}_{\texttt{IN}}$ edges and $\texttt{DATA}_{\texttt{OUT}}$ is driven on negative $\texttt{CLK}_{\texttt{IN}}$ edges. While this also reduces maximum performance it ensures that hold time scales with frequency, which can ensure robust operation via sufficient frequency scaling.

3.2 MBus Logical Design

There are three logical types of MBus nodes: transmitting, receiving, and forwarding. As seen in Figure 3.4, during normal operation, a physical node will progress through each of these logical states. Forwarding is the most common state and is the idle state for MBus nodes. In this state nodes may be completely power-gated and asleep. The only obligation is forwarding $\texttt{DATA}_{\texttt{IN}}$ to $\texttt{DATA}_{\texttt{OUT}}$ and $\texttt{CLK}_{\texttt{IN}}$ to $\texttt{CLK}_{\texttt{OUT}}$. At the start of a new transmission, if a node matches its address, it promotes itself from forwarding to receiving. A node will receive data of arbitrary length until an interjection event ends the transaction. Alternatively, after the first mis-matched address bit a forwarding node transitions to the IGNORE state for the rest of the transaction, where it will remain until an interjection event. To begin transmitting, a node must first arbitrate for the bus. After arbitration is an addressing phase, a data transmission phase, and then an interjection event which is used to end the transmission. The rest of the design explains each of these states and their transitions in detail.

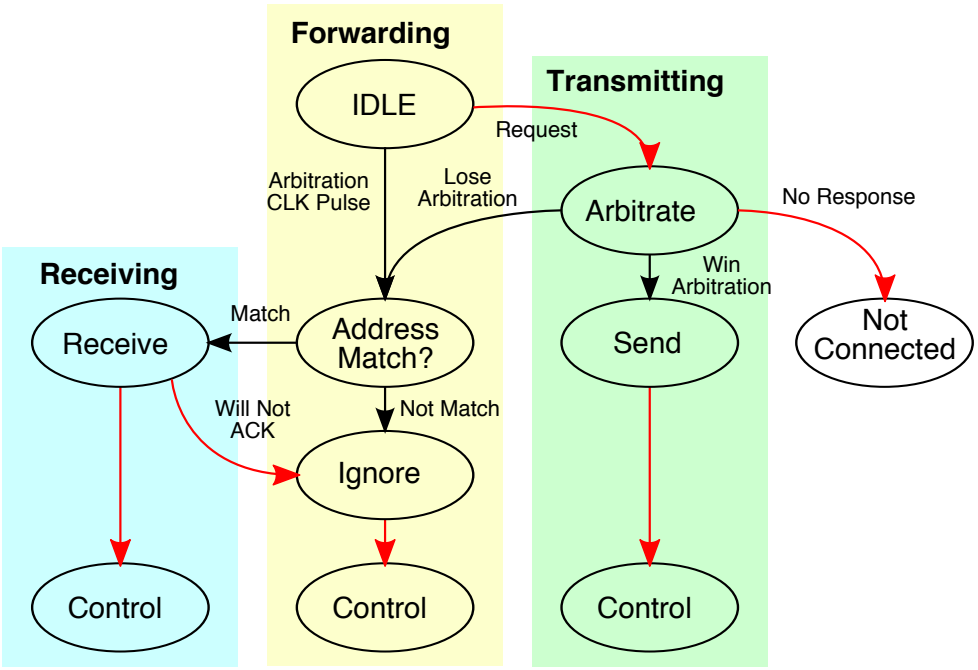


Figure 3.4: **High-level behavior of MBus nodes.** MBus nodes are normally idle. A transaction begins when one or more nodes elect to transmit by starting an arbitration phase (§Arbitration). The winner then transmits a destination address (§Addressing, §Address Assignment), sends payload data (§Data Transmission), and interjects to end the transaction (§Message Termination). Black arrows are synchronous, transitions that match MBus CLK edges, while red arrows are asynchronous. By leveraging the interjection primitive, MBus transmitters can reliably signal the end of message without requiring an embedded length or an out-of-band—extra wire—signal. Not shown are arrows from any state to control due to interjections and from control back to idle.

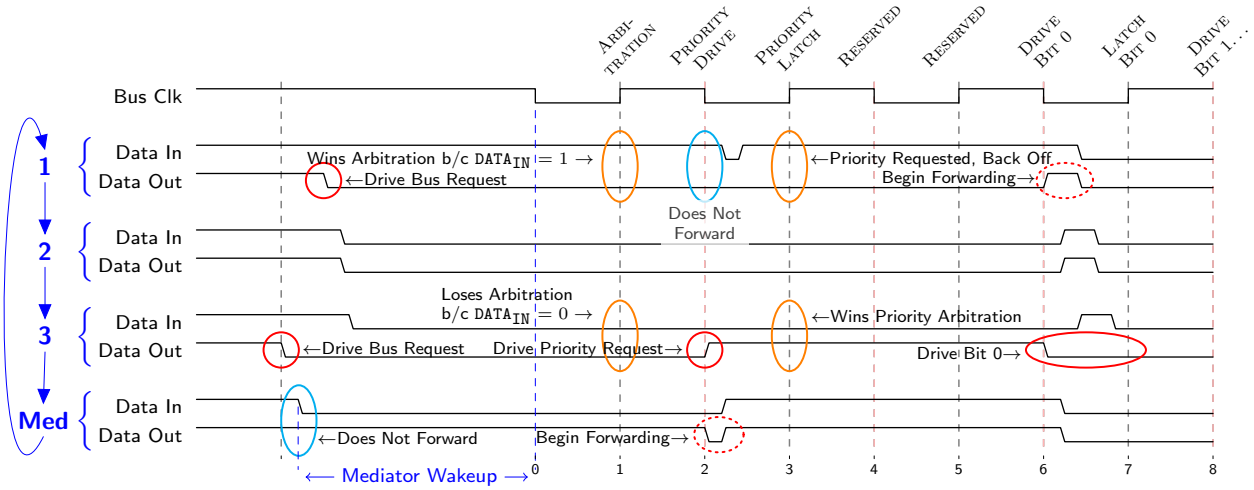


Figure 3.5: **MBus arbitration.** To begin a transaction, one or more nodes pull down on $\texttt{DATA}_{\texttt{OUT}}$. This shows node 1 and node 3 requesting the bus at nearly the same time (node 1 shortly after node 3). Node 1 initially wins arbitration, but node 3 uses the priority arbitration cycle to claim the bus. The propagation delay of the data line between nodes is exaggerated to show the shoot-through nature of MBus. Momentary glitches caused by nodes transitioning from driving to forwarding are resolved before the next rising clock edge.

Arbitration by Mediating Shoot-Through Rings

In the idle state all nodes except the mediator forward CLK and DATA signals around the rings. The mediator drives both its $\texttt{CLK}_{\texttt{OUT}}$ and $\texttt{DATA}_{\texttt{OUT}}$ high in the idle state, which results in all bus lines being high during idle due to forwarding. A node requests the bus by breaking the chain and driving its $\texttt{DATA}_{\texttt{OUT}}$ low. This propagates around the DATA ring until it reaches the mediator, which is not forwarding. The falling edge on $\texttt{DATA}_{\texttt{IN}}$ triggers the mediator self-start. The mediator is given significant leeway in its wakeup duration, for reasons outlined in the next section on power design. During this window, additional nodes may also pull their $\texttt{DATA}_{\texttt{OUT}}$ low, in which case there will be contention for the bus. Once the mediator pulls CLK low, the bus is no longer considered idle.[1]

Arbitration is resolved at the first rising edge of CLK. Any node arbitrating for the bus samples their $\texttt{DATA}_{\texttt{IN}}$ line. If $\texttt{DATA}_{\texttt{IN}}$ is high, the node has won arbitration, otherwise it has lost. This arbitration scheme introduces a topologically-dependent priority on MBus nodes. To afford physically low-priority nodes an opportunity to send low-latency messages, MBus adds a priority arbitration cycle after arbitration. The priority arbitration scheme is similar, except it is the arbitration winner that does not forward DATA and nodes pull $\texttt{DATA}_{\texttt{OUT}}$ high to issue a priority request. Figure 3.5 shows a waveform of arbitration and priority arbitration.

[1]There is a subtle but important point here that idle ends at time 0 not time 1 in Figure 3.5. A node may not pull $\texttt{DATA}_{\texttt{OUT}}$ low after time 0. This requirement follows from the balanced timing design discussed in the previous section, where to be seen reliably by all nodes any change in DATA cannot occur after the falling edge of CLK.

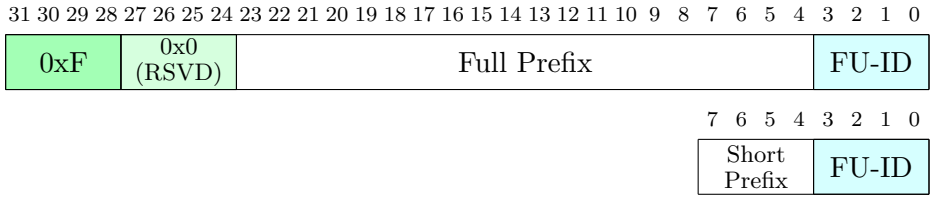


Figure 3.6: **MBus addressing.** MBus addresses are composed of either a full (top) or short (bottom) prefix, followed by a functional unit ID. The short prefix `0xF` is a sentinel that identifies a full address and `0x0` indicates a broadcast message. This allows up to fourteen nodes per instance to be address by their short prefix.

Addressing: Prefixes, FU-IDs, and Broadcast Messages

MBus uses an addressing scheme to direct transmissions. It divides addresses into two components, a *prefix* and a *functional unit ID* (FU-ID). A prefix uniquely addresses a physical MBus interface (one of the actual chips in the system), while FU-IDs are used for higher-level protocol operations. This shares conceptual similarity to I^2C, whose 8-bit addresses are more accurately a 7-bit address and a 1-bit function (read/write). FU-IDs are 4-bits, however, which allows for a richer and extensible functional interface. MBus reserves prefix `0` for broadcast messages. On a shared bus, broadcast messages are cheap to implement in hardware but expensive (linear in system size) to emulate in software, which motivates hardware broadcast support.

Prefix Assignment and (Optional) Enumeration

To retain the efficiency afforded by short addresses while allowing for a diverse ecosystem of unique components, MBus has both 4-bit *short prefixes* and 20-bit *full prefixes*. Figure 3.6 summarizes the whole MBus addressing scheme.

Every chip design is assigned a unique, 20-bit *full prefix*. Full prefixes allow nodes to refer to one another with static addresses at the cost of 16 bits of additional overhead per message. MBus also uses run-time enumeration to assign 4-bit *short prefixes*. Enumeration is a series of broadcast messages containing short prefixes that can be sent by any node (although in practice most likely by a microcontroller). All unassigned nodes attempt to reply with an identification message and the arbitration winner is assigned the enumerated short prefix. A result of this enumeration protocol is that a node's short prefix encodes its topological priority. Enumeration is performed once, when the system is first powered. As an optimization, devices may assign themselves a *static short prefix*, akin to I^2C addressing, so if there are no conflicts (known at system design time) enumeration may be skipped.

The short prefix `0x0` indicates a broadcast message and `0xF` is reserved to indicate full prefixes, which leaves MBus with 14 usable short prefixes per system. Chips may be addressed using either short or full addresses interchangeably. It is sometimes advantageous for a system to have two copies of the same chip (e.g. memory), which requires short prefixes and enumeration to disambiguate.

Data Transmission

Data transmission is simple, synchronous, serial signaling. Recall from Section 3.1 that MBus transmitters drive data on the falling edge of `CLK` and receivers latch data on the rising edge of `CLK`. While standard flops can only be clocked on one edge (rising or falling), only the internal data FIFO needs to be clocked on the falling edge, thus this does not violate the synthesizability requirement.

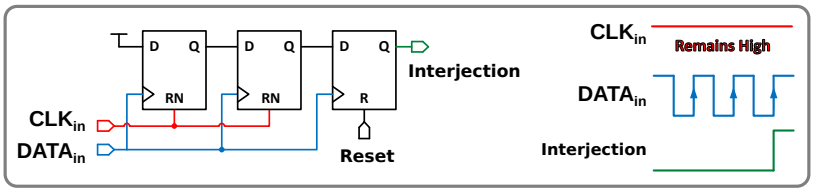
Acknowledgments

At the end of a message, the receiver either ACKs or NAKs the entire message. In MBus, any node may terminate any message at any time, even a forwarder. The transmitter may end the message when it is finished, or the receiver may interject mid-message to indicate error, e.g. buffer overrun. Thus, by not interjecting, a receiver *implicitly* ACKs every byte. This is less powerful than I^2C ACKs, which can detect a dead receiver after the first byte, but is more efficient during error-free operation and allows MBus to scale to long (multi-kilobyte) messages with a fixed, length-independent overhead.

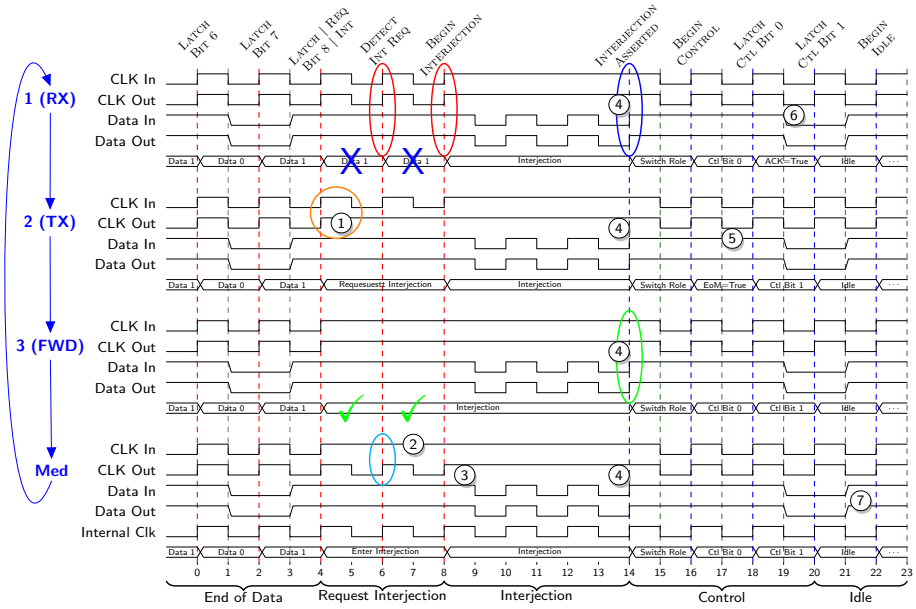
In-line Interjections End Messages, Provide Reliable Reset, and Enable Responsive Messaging

At any point, the bus may be interrupted by an MBus *interjection*. In normal MBus operation, `DATA` never toggles meaningfully without a `CLK` edge. This allows the design of a reliable, independent interjection-detection module as shown in Figure 3.7a, which is essentially a saturating counter clocked by `DATA` and reset by `CLK`. The interjection signal acts as a reset signal to the Bus Controller, clearing its current state and placing it in *control* mode. MBus control is two cycles long and is used to express why the bus was interjected, either an end-of-message that is ACK'd or NAK'd or to express some type of error. Figure 3.7 shows the end of an MBus transaction sent from Node 2 to Node 1 that is ACK'd. Notice that the MBus interjection request mechanism, holding `CLK` high, results in nodes observing a varying number of clock edges. MBus requires that messages be byte-aligned to resolve this potential ambiguity. As a result, a small amount (up to 7 bits) of padding may need to be added to MBus messages.

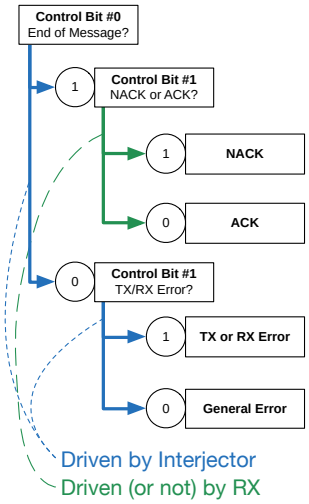
MBus interjections are used both for extreme cases, such as rescuing a hung bus or indicating receiver error, and as a regular end-of-message signal. Any node may generate an interjection at any time. This allows for unambiguous signaling of control functions that can resynchronize a bus without requiring out-of-band signals like chip-selects or a reset line. This further allows a node with a latency-sensitive message to interrupt an active transaction, enabling responsiveness across a diverse array of workloads not possible with current buses.



(a) Interjection detector.



(b) Interjection timing with successful TX and ACK.



(c) Control bits.

1. After the transmitter sends all of its data it requests interjection by not forwarding `CLK`.
2. The mediator detects that a node has stopped forwarding `CLK`.
3. The mediator stops toggling `CLK` and begins toggling `DATA` – the interjection sequence.
4. After interjection, the mediator begins clocking again. Node 1 discards the extra bits because they are not byte-aligned.
5. The transmitter signals a complete message by driving Control Bit 0 high.
6. The receiver ACK's the message by driving Control Bit 1 low.
7. After control, the mediator stops forwarding `DATA`, driving it high, and returning the bus to idle.

(d) Detail of an interjection and control event.

Figure 3.7: **MBus interjection and control.** The MBus interjection sequence provides a reliable in-band reset signal. Any node may request that the mediator interject the bus by holding $\texttt{CLK}_{\texttt{OUT}}$ high. The mediator detects this and generates an interjection by toggling `DATA` while holding `CLK` high. An interjection is always followed by a two-cycle control sequence that defines why the interjection occurred.

Return to Idle

After latching the final Control Bit (time 20 in Figure 3.7b), one final edge (time 22) is generated to enter IDLE. It is necessary for the bus to generate this edge to enable member nodes with no clock the ability to fully participate. If the data line is low at this edge, then it is considered the start of a new arbitration cycle rather than a return to IDLE. The mediator node will pull the clock low again in response and begin a new transaction. With this design, ultra-low power, clock-less nodes can receive, react to, and respond to messages (albeit with possibly tight timing constraints).

3.3 MBus Power Design

MBus aims to support very low power operation. As such, it is expected that systems leveraging MBus may need to support power-gating all or part of the system. This section looks in detail at some of the nuances of very low power design. It discusses the requirements to support power-gated systems, how such systems integrate with MBus, and how power-oblivious chips can seamlessly interact with power-conscious chips, promoting interoperability.

A Brief Background on Power-Gating

In the low-power design space, a simple and important concept is the ability to power-gate, or selectively disable, portions of a system that are idle. Many designs today already support clock-gating, which "freezes" idle subsystems and eliminates dynamic losses. However, there is still a small amount of static leakage present. Smart Dust systems need to reach idle power on the order of single digit nanowatts, which means that static leakage can be a significant factor in the lowest power states. Power-gating literally removes power from idle sections of a chip, which reduces the power draw of idle silicon to effectively zero.[2]

Several challenges with power-gating include: how to preserve state during idle windows, how to connect power-gated modules to other powered or power-gated modules, and how to wake and sleep power-gated modules deterministically. This document does not seek to address all these issues, rather it demonstrates how MBus can help implementations. For a more detailed reference power-gated design with MBus, consult the *MBus M3 Implementation Specification*.

Recall from Section 2.2 that sleeping and waking power-gated modules requires four signals. When powering on, these signals require four sequential steps; when powering off, they require only two sequential steps:

[2]There is, of course, still non-zero loss in the power-gating mechanism itself.

Signal Name	Function	Power-Up	Power-Down
POWER_ON	Controls Power-Gating	1st	2nd
RELEASE_CLK	Supply Clock to Internal Logic	2nd	2nd
RELEASE_ISO	Electrically Isolate Module I/O	3rd	1st
RELEASE_RST	(De)Assert Reset	4th	2nd

Transparent and Efficient Hierarchical Wakeup

The key insight that enables MBus's power-oblivious properties is that a power-gated node can use the edges on the CLK line from arbitration as stable, predictable pulses to drive power-on circuitry. At the start of a transaction, a node is either already powered and actively participates in arbitration, or it uses the arbitration edges to drive its wakeup sequence. Thus, a node powers-on in the middle of its state machine, when it has just "lost" arbitration and is then ready to listen for the incoming address.

To avoid powering on all of the nodes in a system whenever any two nodes communicate, the MBus design allows for hierarchical wakeup. Conceptually, MBus imagines a small "Bus Controller" block, which is responsible only for managing MBus operation for a node. This Bus Controller requires no dedicated clock source and can operate solely off of the MBus CLK. This is critical as it allows the node to remain low impact when its bus logic is partially powered due to an active bus transaction (so that it can know when the bus is free) that the node is otherwise not participating in. The Bus Controller is then responsible for powering on the next block, which MBus calls the "Layer Controller", when necessary. In practice, power-optimized MBus designs will not wake the rest of the node until an address match. This design allows any node to transmit to any other node at any time while ensuring that the destination node, and only the destination node, will be powered on to receive the message.

Waking the Bus Controller

To receive an MBus transmission the power-gated node's Bus Controller must first be activated. Referring to edges from Figure 3.5, edges 1, 2, 3, and 4 provide the required signals. Mapping power edges to MBus protocol edges:[3]

Arbitration → POWER_ON
Priority Drive → RELEASE_CLK
Priority Latch → RELEASE_ISO
Reserved → RELEASE_RST

[3]Historical note: The original MBus design did not include the Reserved cycle (times 4 and 5 in Figure 3.5). These Reserved edges are not required for power correctness (without them, the Drive Bit 0 edge would map to RELEASE_RST and the node's Bus Controller would be active by Latch Bit 0 as needed). The first RTL implementation of MBus had a small error that introduced an extra cycle between regular and priority arbitration. Upon discovery and discussion, the team decided that preserving one cycle of overhead for future extensibility was a good tradeoff but did correct the implementation to place the two arbitration cycles next to one another in time. In this way, the Reserved cycle is available regardless of initial power state.

In practice, most Bus Controller implementations will not require the RELEASE_CLK signal as the MBus clock is (by definition) sufficient for all bus operations, however it is included in consideration for designs that may require it. A Bus Controller that is awoken using MBus edges will find its first rising clock edge to be the Reserved cycle.

Waking the Layer Controller

If the node matches the destination address, it must wake whatever it is attached next up the chain, namely the Layer Controller. This means the clock-less Bus Controller module must harvest clock edges from MBus to generate the power control signals to power on the layer. While there is no gap between address transmission and data, the Bus Controller can dual-purpose the first few bits of data. Recall that because of the 2-bit transmission ambiguity, MBus data is strictly byte-oriented. This means that Bus Controller designs can safely buffer up to one byte of data before they need to hand the data to the rest of the node. Collecting eight bits of buffered data allows for ample CLK edges to drive wakeup for the rest of the node.[4]

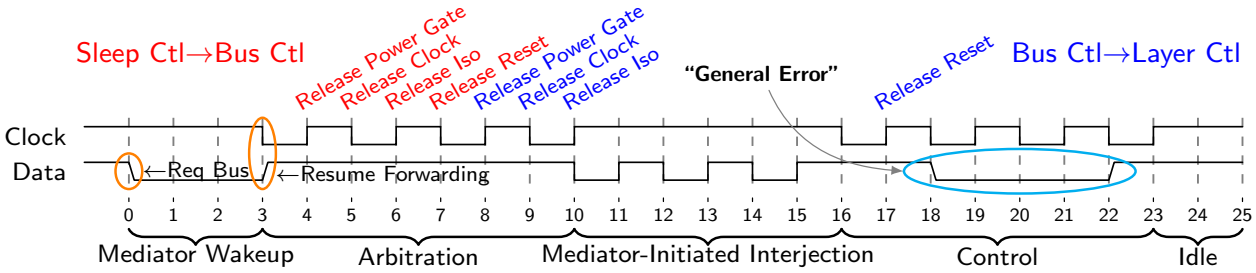
Optimizing the Bus Controller

If the node does not match the destination address, its only responsibility is to forward CLK and DATA signals. However, the node also must track when the transaction completes. This is another consideration and advantage of the MBus interjection design, which allows Bus Controllers to gate their clock tree during data transmission. Only the three-flop interjection detector needs to be clocked to allow a node to rejoin during the Control cycle.

Supporting Intra-Node Wakeups

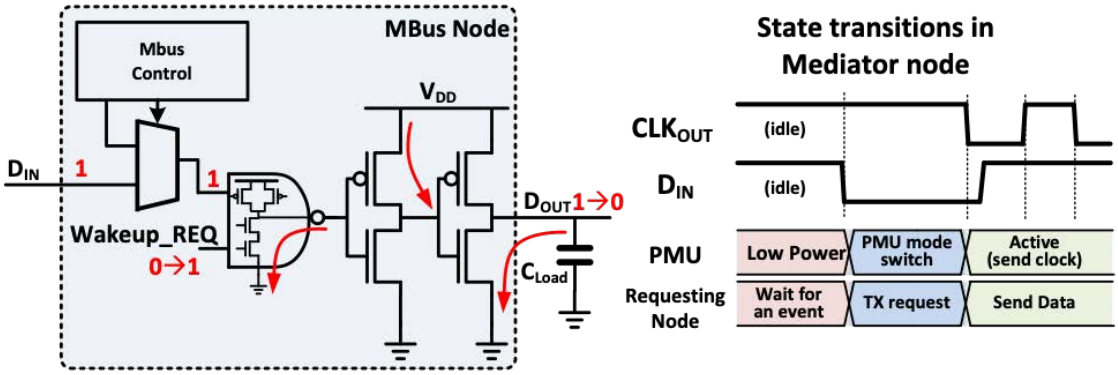
Partially power-gated nodes will transparently wake up to receive messages, but they may also wish to voluntarily wake themselves, usually in response to a locally generated event from, say, a timer or sensor. For instance, a node with an always-on analog circuit (e.g. an ultra-low power motion detector embedded in an imager, as described in Section 5.3) may wish to wake the digital logic on the rest of the node to take a picture or send an alert message. The always-on MBus frontend provides a simple interrupt port that the component can assert. Upon interrupt request, the frontend will generate a null message, as shown in Figure 3.8. This null message causes the mediator to generate the clock edges needed to wake the rest of the node. With this design, a power-conscious node can leverage all of its power-saving faculties without requiring support from any other system component, simultaneously maximizing interoperability and efficiency.

[4] One design point explicitly required by MBus is the acknowledgment of zero-length messages. Depending on application, a node may not require awakening for a zero-length message. Due to the nature of the MBus interjection procedure, however, as many as two bits may be received that will be discarded (Figure 3.7). A Bus Controller design that attempts to minimize wakeups should therefore not begin the wakeup process until latching the 3rd data bit.



(a) Power-gated nodes repurpose CLK edges to drive well-timed power-on signals. This waveform shows the self-wake, where a power-gated node uses a simple pull-down signal on DATA to trigger the mediator to first prepare the system for higher power operation if needed (see (b)) and then to to begin generating CLK edge the node can use to wake itself. In a null transaction (shown here) the power-gated node must resume forwarding† DATA before the arbitration edge so that the mediator can detect an "error" of no arbitration winner. In response, the mediator will raise a general error, and return the bus to idle. The null transaction produces enough edges to wake all of the MBus hierarchical power domains in a manner that is transparent to non-power-aware devices.

†Note that a node cannot simply drive DATA high here, as it is possible that another node may also be arbitrating for the bus at this time. A node intending a self-wake procedure must be prepared for the case where an actual bus transaction takes place as well. In the common case, forwarding will cause DATA to go high again, as the mediator always drives its DATA_{OUT} high during the initial arbitration cycle, and it is the only node not forwarding in the ring.



(b) Some power management unit designs include more efficient low-power modes with limited dynamic range for idle periods. In addition to clock-free operation, this is a second reason for the MBus wakeup signal to be a simple signal, as this draws negligible power and the power management unit can then prepare the system for higher-power operation before allowing the mediator logic to begin bus events, which will wake more sub-circuits and draw more power.

Figure 3.8: **MBus wakeup.** Instead of requiring every node to include custom cold-boot circuitry, MBus provides a mechanism to wake a node in response to a single falling edge. The design further considers system-wide power management, with affordance for regulator intervention and update before allowing system components to power on.

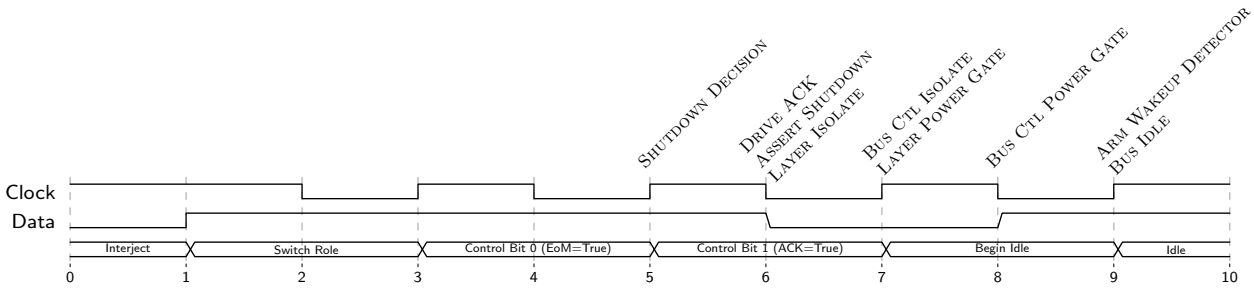


Figure 3.9: **Shutdown timing.** The shutdown command is not confirmed until time 5 when the transmitter indicates a valid End of Message signal. At time 6, the Bus Controller acknowledges shutdown, asserts the `SHTDWN` signal to the Sleep Controller, and isolates the Layer Controller. At time 7, the Sleep Controller isolates the Bus Controller, and isolating the Bus Controller by definition power gates the Layer Controller. At time 8, the Sleep Controller power-gates the Bus Controller, completing shutdown. At time 9 the bus is idle, and the Sleep Controller is waiting for the next wakeup.

Re-Asserting Power-Gating

MBus edges can also be harvested to drive the control circuitry needed to return both the layer and the Bus Controller to their lowest power states. Figure 3.9 demonstrates how the control edges following the End of Message bit may be used to progressively power-gate first the Layer and then the Bus Controller. Note that Figure 3.9 delays the shutdown procedure until the transmitter asserts End of Message, which indicates that the shutdown message was not in error. This shutdown procedure has the advantage that it allows power management to remain fully decentralized. A node can shut down without any intervention or knowledge of the mediator node. Furthermore, a node with no local clock or self-timed power management circuitry can use bus edges to drive its power-off sequence. This is the final piece required to ensure that the interconnect fully supports the lowest-power components, those with no local oscillators which power-gate aggressively.

3.4 Summary

This chapter has described the design of MBus, a new interconnect for designed to support composable deisgn for the millimeter-scale computing class. MBus introduces a novel "shoot-through" ring topology, which enables energy-efficent, scalable composition. The MBus design also demonstrates the necessity and value of a "power aware" interconnect when composing pitch black silicon and shows how offloading power management into the interconnect can alleviate complex circuits and systems challenges. Thus far, MBus describes only how to reliably send bits between chips. The next chapter will show how assigning structure and semantic meaning to bus messages can empower richer and more advanced system synthesis.

Chapter 4

Building Modular Components

The MBus interface describes how to physically connect chips and how to transmit bits of data between them. This chapter discusses what those bits of data should look like. It also argues for the standardization of core building blocks, namely registers and memories, and both their internal and external interfaces. For chip designers, this standardization provides scaffolding on which to build new designs, which can accelerate design and reduce risk. For system designers, this additional standardization of component design allows for predictability and opens new opportunities for decentralization and automation.

4.1 A Standard Scaffolding

Much like MBus described a standard for composing chips in a system, this section looks at how to compose the modules that make up chips. In particular, this looks at the glue logic that holds the basic components of chips together. The goal of this scaffolding is to ease and accelerate the integration of new circuits (e.g. a faster or more energy efficient capacitive digital converter [15]) into systems. The key observation is that much of the machinery that connects interesting frontends to systems is near-identical already. Indeed, many design houses have their own internal semi-standard core logic block that is reused across designs. If instead the interface were to provide this scaffolding, it could then impose structure on otherwise arbitrary decisions (e.g. "which register address turns this chip on and off?"). With core operations standardized, system-level behavior can be encoded independent of specific instantiations. This allows for re-use of standard libraries in the software controlling such systems or the integration of possibly complex chains of events into hardware.

Fundamentals of Components

What makes up a modern chip? From an operational standpoint, nearly every design can be reduced to some subset of the following:

i. Configuration data – Small amounts of data that affect behavior (e.g. sampling resolution); this data can be transient or persistent.

ii. Large-volume data – One or more "memory-like" storage systems; these may be persistent (e.g. flash or FRAM) or transient (e.g. SRAM) and with other tradeoffs (e.g. energy or latency), but ultimately can be modeled as readable and writable data in a linear address space.

iii. Synchronous logic – Anything from a simple state machine to a full-blown microcontroller; this is anything with a local clock and a predictable, synchronous interface.

iv. Asynchronous logic – Historically this would be analog sensors, but this is now expanding to ideas such as analog compute accelerators; this is anything without a local clock that needs interface logic to connect to a synchronous interface.

v. Event interface – Some mechanism for either triggering the beginning of a hardware event or being alerted that one has completed.

Learning from the pre-SoC Era

When Hill et al. first described the architecture of motes, which were the resource-constrained computing class of the turn of the century, the design concerns centered around how to "provide *efficient modularity*" on devices that "are *concurrency intensive*" [63]. At the time, centimeter-scale modules were similar in capability to millimeter-scale devices today. The core of mote designs was a microcontroller (MCU) with a central processing unit, some on-board memory, and a number of I/O interfaces. Across a mixture of general purpose I/O (GPIO), UART, SPI, and I^2C, various peripherals (e.g. radios, sensors, memories) were attached to the MCU. Generally, none of these peripherals were capable of independent operation. This meant the MCU had to control each peripheral. In practice, this drove the concurrency intensive requirement, as operating multiple peripherals at once required carefully interleaving controls from the single central processor while meeting the timing requirements of each peripheral.

In contrast, today many peripherals include local processing. Designs now expect higher-level interfaces from remote modules that encapsulate timing-sensitive operations. As general purpose processing continues to become cheaper, this trend of pushing responsibilities onto peripherals has continued. Indeed, many radio "peripheral" ICs include full general-purpose processors that can be reprogrammed to execute arbitrary application code. However, general purpose compute is less area and energy efficient that fixed-function controllers.[1] Millimeter-scale systems simply do not have the floorplan area, let alone the surplus energy, to put arbitrary compute cores on every chip. This does not mean, however, that these devices

[1]While "resource-constrained" for Smart Dust concerns itself with physically small computing devices, it is worth observing that this is the same reality that is driving the resurgence in accelerators and domain specific architectures in macro-scale computing.

should return to bare, low-level control interfaces. To allow robust, modular integration, it is important that timing-sensitive operations be encapsulated in modules as much as possible.

Learning from SoCs

While systems on chips (SoCs) do result in a monolithic integrated circuit, internally their design is still modular. Most SoC designs center around a bus, to which processing, memory, and peripherals are attached. As these systems are integrated on a single die, their interfaces do not share the same I/O constraints as an interchip bus like MBus. This allows SoC interconnects to be wide buses (generally 32-bits in modern designs).

Modern microcontrollers have shown that memory-mapped I/O (MMIO), coupled with an interrupt mechanism, can act as a singular interface to abstract all of the components of chips. Under the hood, hardware implementations can attach systems of varying complexity to this generic 'address plus data' interface. For the data-oriented components (i, ii), a direct mapping to the underlying data stores is often sufficient. For logics (iii, iv), a small interface state machine can be written. At the software level, then all hardware can now be accessed through this singular interface.

Internally, there is often more than one actual memory bus, however. While this is in part for physical scalability (limiting the load on the shared bus lines), there is also an architectural tradeoff between the complexity and the performance of various memory bus designs. For this reason, the ARM Advanced Microcontroller Bus Architecture (AMBA) includes both an AMBA High-performance Bus (AHB) and an Advanced Peripheral Bus (APB) [64]. The AHB is well-suited to "memory-like" structures (ii). It supports efficient reads and writes of large volumes of data, at the cost of a more complex interface state machine for components that attach to the bus. The APB is optimized for simpler "register-like" structures (i). It is easier to interface with, at the cost of higher protocol overhead and latency for repeated transactions. The TileLink specification from SiFive similarly allows for a mixture of 'lightweight' and 'heavyweight' peripherals [65]. The takeaway from SoC interconnects design then is that it is reasonable to bin components, and the interfaces to attach them, into two tiers of capability to balance integration complexity and performance.

Learning from x86

Some systems split I/O and memory into isolated address spaces. In x86, interfacing for simpler peripherals was originally done via port-mapped I/O (PMIO), which is also sometimes referred to as direct I/O mapping. The x86 instruction set includes two classes of load and store operations, those that operate on main memory (e.g. `LEA`) and those that operate on peripherals (e.g. `OUTW`). Further, x86 also specifies where in the I/O address space certain peripherals will be placed. For example, the 'first serial port' will be at address `0x3F8`.

Isolating peripherals to their own address space easily allows for separate interfaces for peripherals and memories. Again, this is valuable as it allows for less complex (though less performant) peripheral interfacing. However, modern hierarchical interconnects such as

AMBA and TileLink demonstrate that address space separation is not required to achieve separate interfaces for components of varying complexity and capability.

On the surface, PMIO also provides runtime separation of transactions. In its simplest form, MMIO would serve its entire address space from a single, physical bus. In that case, a transaction with a slow peripheral could block the central bus for many cycles, effectively pausing the whole system. This is similar to the performance bottleneck exhibited by pure von Neumann machines, when data and instruction accesses conflict with one another. The Harvard architecture mitigates this contention problem by servicing different regions of the address space by different physical buses. The same solution is available to the MMIO interface, which can present software with a unified address space that is internally partitioned to multiple physical buses. Indeed, the opposite is even possible. PMIO purports two separate address spaces, but the underlying architecture could route accesses to a single physical bus if desired. For the hardware designer, PMIO and MMIO are interchangeable.

The unified address space of MMIO does sometimes present challenges, however. One of the most (in)famous examples is the "PCI Hole" or the "3 GB wall", catch-all terms coined around the time that consumer PCs began to exceed 2 GB of RAM. At the time, 32-bit CPUs were still common, which in principle should have been able to access as much as 4 GB of RAM. In practice, however, motherboards often laid out peripherals sparsely across the top 1 GB of the address space. This meant that even though large volume memory modules were available, systems could not take advantage of them. While techniques such as Physical Address Extension eventually allowed access to more memory, it is important to consider the long-term implications of allowing immutable hardware to claim regions of an address space.[2]

Learning from IDE, ATA, DMA, and PPI

The earliest ATA bus (then-named just IDE) supported only the "PIO 0" transfer mode ("Programmed I/O"). PIO designs look much like pre-SoC designs, they require the central CPU to continually manage data transfers. Once set up, however, larger transactions are highly repetitive events that require comparatively little logic to operate. This insight is the

[2]As some fun history, sometimes MMIO assumptions can provide opportunity as well. The original IBM PC used the Intel 8088 CPU, which had 20 address lines and was thus capable of addressing 1 MB of memory. Of this, the bottom 640 KB (which perhaps 'ought to be enough for anyone') was available as traditional RAM, while the top 384 KB were reserved for graphics, peripherals, and the BIOS. To extend system memory then, external memory controllers (with sometimes several MB of memory) would claim a small region of the peripheral address space. This peripheral was actually a paged memory controller, which would allow *processes*—which at the time were allowed direct peripheral hardware access—with an understanding of the memory controller interface the ability to manage vast amounts of additional memory for their own use through a 64 KB paging window. This approach required no modification to the MS-DOS operating system, which for a long time did not have a sense of the machine having more than the original 640 KB of memory. Processes that used several MB of memory simply seemed to perform a large number of peripheral transactions. In the end, MS-DOS memory segments would include RAM, ROM, Conventional, Reserved, Expanded, Enhanced Expanded, Extended, and High memory areas, many of which hid in similar limited windows. For those interested in more details, I recommend Jerry Crow's SIGICE writeup [66].

foundation of DMA (Direct Memory Access), which frees the CPU. In traditional computing paradigms, DMA allows for high performance, as DMA enables parallelism and allows the CPU to continue other processing. For the resource-constrained ecosystem, however, the value is again in resource reduction – with DMA, only the DMA controller and memory need to be powered during a transaction.

Newer SoCs take this concept a step further. Nordic Semiconductor provides the Programmable Peripheral Interconnect (PPI), Texas Instruments provides MicroDMA (µDMA), and other vendors are adding similar constructs. One could think of a traditional DMA engine as a programmable compute unit that handles a "send-byte" operation and then evaluates to either run the "send-byte" operation again or to run a "notify-complete" operation. These new designs add a richer, but still limited set of capabilities. Additional actions include operations such as "assert GPIO pin". Conceptually, this is further extension of the DMA principle: pre-programming rules to free the CPU from operation. The design of these systems, however, is such that only a limited surface area of peripheral capabilities are exposed by these interconnects. A PPI event can only trigger certain, dedicated PPI tasks. Other peripheral tasks require CPU intervention and MMIO writes.

Exporting MMIO as the System Interface

In MPQ, the insight is to exploit the generality of memory and MMIO interfaces. The key design points are that each chip's DMA controller is itself controlled by (possibly remote) MMIO operations and that any chip can trigger a DMA transaction between any other chips, possibly not including itself. In this way, a CPU chip, with its large memory stores, can write an arbitrarily complicated sequence of pre-programmed events. To chain events, the end of each DMA transaction is a series of memory writes that configure another DMA controller to issue a new transaction. Because everything is exposed via MMIO, this design maximizes generality. Any chip and any peripheral can trigger any operation on any other chip.

Implications on Reliability, Trust, and Security

There is a natural tension between modularity and efficiency. With tightly integrated SoCs, it is readily accepted that a catastrophic failure (or malicious design) in one of the integrated peripheral modules could compromise the system. This design-point implies SoC-like trust among integrated chips. In many ways, this is the norm in embedded designs however. Malicious peripherals could always "take-down" a system with unending interrupt storms, denial-of-service on shared interconnect buses like I^2C, or simply consuming energy at an unsustainable rate. One might imagine chips that add various forms of protected memory regions to help mitigate these risks. Genuine distrust of a tightly integrated peripheral, however, requires significantly more platform support [67].

Handling Asynchronous Events

Events (v) require some form of interrupt mechanism. However, even this interrupt mechanism can be integrated with the memory abstraction. Cores can define memory addresses that map to events—commonly referred to as interrupt vector tables—which execution will jump to when an event occurs. Alternatively, a single interrupt execution path may exist, at which point one or more MMIO addresses can be queried to establish which event occurred.

One to Any and One to Many

While an MMIO abstraction enables generality, it can lose some of the advantages of standardization first highlighted in this section. Consider an emergency situation for a sensor node: a brown-out detector is tripping and to prevent power collapse, every node that can power down must do so. For this reason, the bus itself defines a limited broadcast protocol to express system events. These are the types of things that are required to support operation (e.g. address enumeration) as well as what is reasonably generic across all systems (e.g. basic power management). Then there is the question of how to efficiently export a DMA-optimized MMIO-like abstraction across chips. These two concerns make up the rest of this chapter: first, how to build system-level abstractions, and then, how to design a distributed MMIO interface.

4.2 MBus Broadcast Protocol Design

Section 2.2 observes that for broadcast messages to be effective and reliable for system designers, every node in the system has to opt-in. For this reason, the broadcast protocol described here is included as part of the MBus specification. While some messages, such as enumeration, have to do with the operation of MBus, others, such as power management, have more to do with system operation.

Recall the addressing in MBus is normally divided into a prefix and a function unit ID (FU-ID). The prefix `0x0` is used to indicate a broadcast message. For broadcast messages, the FU-IDs are used as "channel" identifiers. The MBus specification reserves channels 0-7 (MSB `0`) for operations that should be standardized across all systems. This standardization allows both chip designers and system designers to rely on ecosystem-wide operations. As an example, for energy harvesting systems, when energy reserves are low, the energy management chip can send a standard "energy critical" message that all chips in all designs are able to respond to in a meaningful way.

Broadcast Messages (Address `0x0X`, `0xf000000X`)

MBus defines the broadcast short prefix as `0b0000` and the broadcast full prefix as `0x00000`. Broadcast messages are permitted to be of arbitrary length. Messages longer than 32 bits may be silently dropped by nodes with small buffers. A node **must not** interject a broadcast

message to indicate buffer overflow. Interjections are permitted for broadcast messages greater than 4 bytes in length.

For broadcast messages, the functional unit ID field is used to define broadcast *channels*. Broadcast channel selection is used to differentiate between the different types of broadcast messages. MBus reserves half of these channels and leaves the rest as implementation-defined. The MSB of the broadcast channel identifier (address bit 3) shall identify MBus broadcast operations. If the MSB is `0` it indicates an official MBus broadcast message as specified in this document and subsequent revisions. Broadcast messages with a channel MSB of `1` are implementation-defined. It is recommended that nodes leveraging implementation-defined broadcast channels provide a mechanism to dynamically select broadcast channel to help mitigate conflicts.

A broadcast message that is not understood **must** be completely ignored. During acknowledgment, an ignorant node shall forward.

Broadcast Messages and Power-Gating

Some systems may have inter-node dependencies on `Layer` power state, e.g. activating a higher power regulator before a high-power component. For this reason, by default nodes must not change `Layer` power state upon receipt of a broadcast message, excepting messages that explicitly change `Layer` power state.

Some nodes, for example a general purpose processor that is snooping, may want to wake their `Layer` for all messages. This is permitted, but nodes with non-standard broadcast power behavior must clearly document power semantics to be MBus compliant.

Broadcast Channels and Messages

This section breaks down all of the defined MBus broadcast channels and messages. All undefined channels are reserved and shall not be used. A node receiving a broadcast message for a reserved channel shall ignore the message. It **must not** acknowledge a message on a reserved channel and **must** forward during the acknowledgment cycle.

All MBus broadcast messages, except those sent on Broadcast Channels 2-7: Reserved, follow a common template. The messages are 32 bits long. The four most significant bits identify the message type/command. Some messages do not require all 32 bits. The unused bits are named *insignificant bits*. Messages may be truncated, omitting the insignificant bits on the wire.[3]

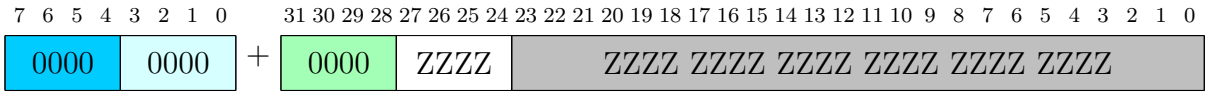
All examples are shown with short addresses for space. There is no distinction between the use of the short or full broadcast address. Bitfields are presented **Address** + **Data**. Addresses are broken down into the Broadcast Prefix and the Broadcast Channel. Data is broken down into a Message Type Specifier and the message itself. In the bitfields, `0` and `1` indicate bits that must be set to that value, `X` indicates bits that depend on the

[3]With the caveat that all MBus messages must be byte-aligned. Some insignificant bits may still be sent on the wire as a consequence.

current message, and `Z` indicates bits that should be *ignored*—accept any value, send as `0`. Insignificant Bits are also indicated as `Z`.

Broadcast Channel 0: Node Discovery and Enumeration Channel 0 is used for messages related to node discovery and enumeration. Channel 0 messages either require a response or are a response. Channel 0 response messages should not be sent unless solicited. The `Bus Controller` is responsible for handling Channel 0 messages. Channel 0 messages should not affect `Layer` power state.

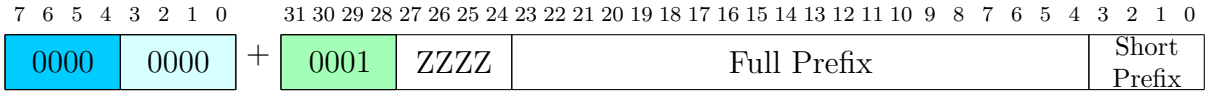
Query Devices



The query devices command is a request for all devices to broadcast their static full prefix and currently assigned short prefix on the bus. Every MBus node must prepare a Query/Enumerate Response when this message is received.

All nodes are required to support this message and respond.

Query/Enumerate Response



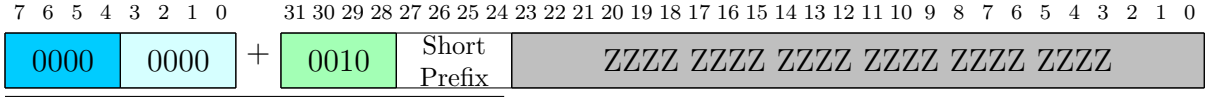
This message is sent in response to a Query Devices or Invalidate Prefix request. When responding to Query Devices, every node will be transmitting their address, and nodes should anticipate losing arbitration several times before they are able to send their response.

The top four bits of the data field identify the message as a Query Response. The next four bits are ignored. The following 20 bits contain the full prefix of the node. The final 4 bits are the currently assigned short prefix. Nodes that have not been enumerated should report a short prefix of `0b1111`.

This message must be sent in response to Query Devices or Enumerate Node. When responding to Query Devices, nodes **must** retry until the message is sent. When responding to Enumerate Node, nodes **must not** retry sending if arbitration is lost and **must** retry sending if interjected.[4]

All nodes are required to support this message.

Enumerate Node



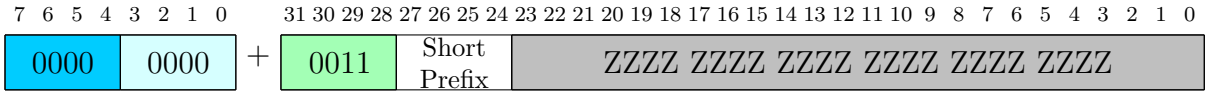
[4] An interjection should not occur during this message. Such an interjection would be an error.

This message assigns a short prefix to a device. All nodes that receive this message and do not have an assigned short prefix **must** attempt to reply with a Query/Enumerate Response. Nodes with a short prefix shall ignore the message (broadcast NAK, that is, forward). Nodes shall perform exactly one attempt to reply to this message. The node that wins arbitration shall be assigned the short prefix from this message. Nodes that lose arbitration shall remain unchanged.

Nodes that have an assigned short prefix shall ignore this message.

All nodes are required to support this message and respond if appropriate.

Invalidate Prefix



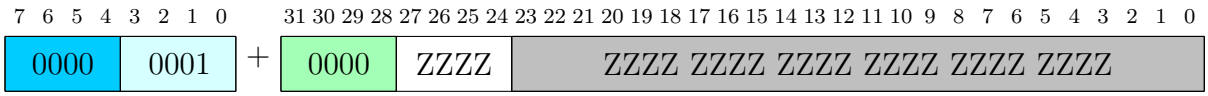
This message clears the assignment of a short prefix. The bottom 4 bits specify the node whose prefix shall be reset. A node shall reset its prefix to Unassigned Short Prefix in the MBus Specification. If the prefix to clear is set to Unassigned Short Prefix, then all nodes shall reset their prefixes.

All nodes are required to support this message.

Broadcast Channel 1: `Layer` Power Channel 1 is used to query and command the `Layer` power state of MBus nodes. Power-oblivious nodes may ignore channel 1. Power-aware nodes whose power model does not align well with these commands may ignore channel 1 messages *except* All Sleep. All nodes capable of entering a low-power state **must** enter their lowest power state in response to an All Sleep message.

A node `Layer` is implicitly waked when a message is addressed to it, explicitly issuing a wake command is unnecessary to communicate with a node. Nodes may build finer-grained power constructs beyond the macro `Layer` control provided by MBus. For the purposes of MBus, a node's "sleep" state should be a minimal power state. Nodes may have different sleep configurations, e.g. different interrupts that are armed.

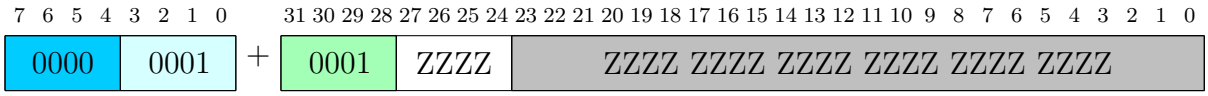
All Sleep



All nodes receiving this message **must** immediately enter their lowest possible power state. The bottom 28 bits of this message are reserved and should be *ignored*.

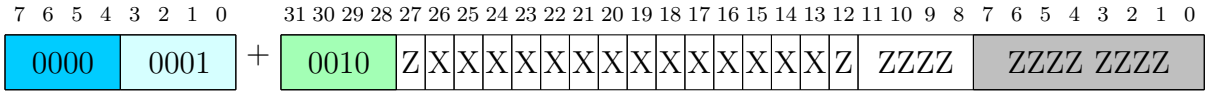
All power-aware nodes are required to support this message.

All Wake



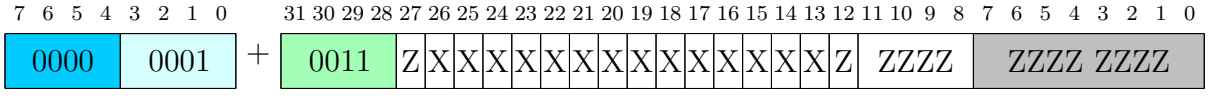
All nodes receiving this message **must** immediately wake up. The bottom 28 bits of this message are reserved and should be *ignored*.

Selective Sleep By Short Prefix



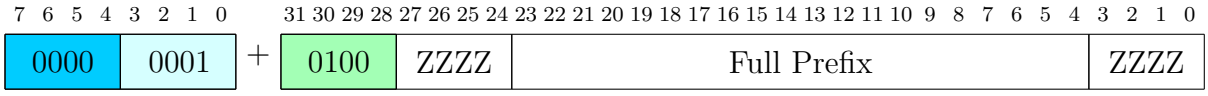
This message instructs selected nodes to sleep. The 16 bits of data are treated as a bit vector, mapping short prefixes to bit indicies. That is, the node with short prefix `0b1101` is controlled by the second bit received (bit 25 in the bit vector above). If a bit is set to `1`, the selected node **must** enter sleep mode. If a bit is set to `0`, the selected node should not change power state. The bits for prefixes `0b1111` and `0b0000` are ignored.

Selective Wake By Short Prefix



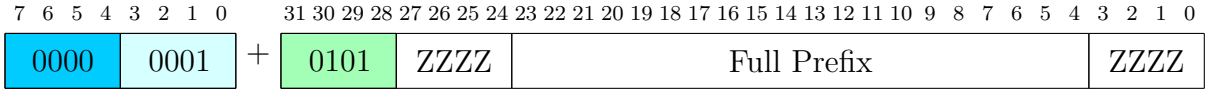
This message instructs selected nodes to wake. The 16 bits of data are treated as a bit vector, mapping short prefixes to bit indicies. That is, the node with short prefix `0b1101` is controlled by the second bit received (bit 18 in the bit vector above). If a bit is set to `1`, the selected node **must** wake up. If a bit is set to `0`, the selected node should not change power state. The bits for prefixes `0b1111` and `0b0000` are ignored.

Selective Sleep By Full Prefix



This message instructs selected nodes to sleep. Any node whose full prefix matches **must** enter sleep.

Selective Wake By Full Prefix



This message instructs selected nodes to wake. Any node whose full prefix matches **must** wake up.

Broadcast Channels 2-7: Reserved

4.3 MPQ: Point-to-Point Message Protocol

MBus also defines a common point-to-point messaging protocol: MPQ.[5] This protocol provides the system-wide MMIO abstraction. MPQ defines two classes of data: register data and memory data. MPQ registers have 8 bit addresses and are 24 bits wide. MPQ memory has 32 bit addresses and stores data that is 32 bits wide. The register address space and memory address space are separate constructs, that is register address `0x00` need not map to memory address `0x00000000`, although aliasing is permitted.

MPQ Registers (Reg #192–255)

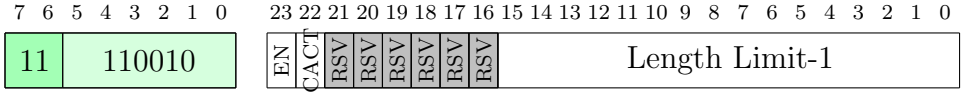
MPQ reserves the top 64 registers for control and configuration. Figure 4.1 gives a visual summary of these registers. This space configures various MPQ capabilities. Much of the space is currently reserved for future expansion. Attempts to write configuration for unsupported features (e.g. configuration of a memory streaming channel on a device with no memory) is undefined. Attempts to read unsupported features **must** NAK or return all `0`. This section documents each of the registers. The top two bits of each MPQ register address are `11` followed by six bits that identify the type. All reserved bits should be treated as RZWI. The MPQ controller stores state about its current operation in the status registers, #240–247. Each MPQ operation describes what the MPQ controller will **Record** and when it will trigger an interrupt. Reg #220: MPQ Record and Interrupt Control controls what events generate interrupts.

Reserved Registers

All registers not specified here are reserved. Writes to reserved registers should be ignored. Reads from reserved registers should return all `0`.

[5] For those chips kind enough to mind their P's and Q's on MBus.

Reg #216: Bulk Memory Message Control



Default: `0x800000`, `0x000000` if no memory.

This register controls the response of this chip upon the receipt of a Memory Bulk Write command.

- {`R[23]`}: `Enable (EN).`
  - Controls whether bulk memory transactions written to this device are enabled. If `EN` is `0`, a node must not modify the contents of memory in response to a bulk memory transaction.
  - **Acknowledgment/Interjection:** The behavior of a bulk write when `EN` is `0` is undefined. Receivers are encouraged to interject and indicate receiver error, however they may exhibit any behavior, including ACK'ing the transaction and silently ignoring it.
- {`R[22]`}: `Control Active (CACT).`
  - If this bit is high, this register's length field acts as a limit for the maximum permitted bulk message length. A bulk message is allowed to write until this message limit is reached. If more data comes, the message **must** be NAK'd. The receiver should interject with receiver error as soon as it knows the length limit has been exceeded.
- {`R[15:0]`}: `Length Limit.`
  - The maximum permitted length in words of a memory bulk write message to any address.

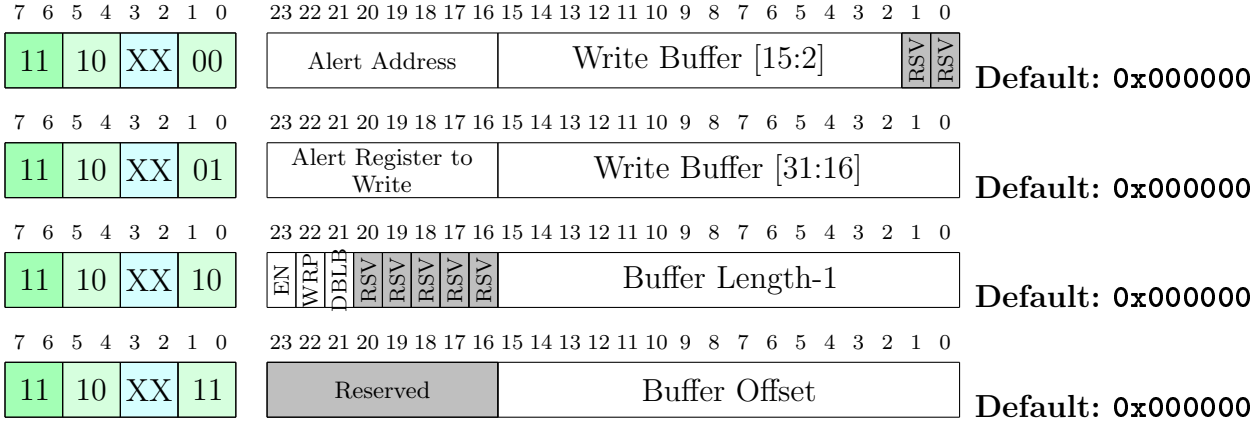
Reg #220: MPQ Record and Interrupt Control

Default: `0x0000XX`

Note: The default is implementation dependent. Most chips default to all interrupts off (`0x000000`).

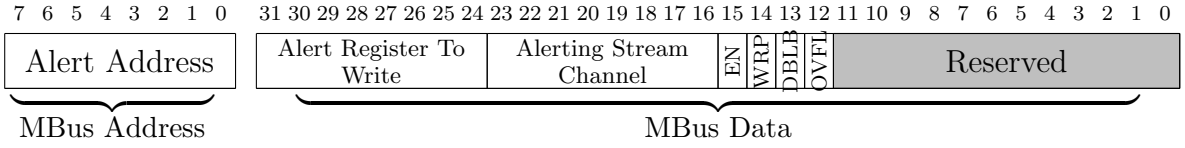
Each time a node completes a MPQ command, it checks the associated command `INT` bit. If high, the layer owner is interrupted.

Reg #224–239: Memory Stream Configuration



In MPQ each node has up to four independent, identical memory streaming channels. Each channel has two configuration registers. The two registers work together to configure each channel.

- {`R01[15:0]`, `R00[15:2]`, `2'b00`}: `Write Buffer`.
  - Pointer to the beginning of a buffer in memory.
- {`R00[23:16]`}: `Alert Address`.
  - Defines where alerts are sent. If the alert prefix (bits 23:20) are set to the full-prefix indicator `1111`, the alert is suppressed.
  - Alerts are sent whenever the end of the buffer is reached. If `DBLB` is active, an alert is also sent when the halfway point of the buffer is reached.
  - When a memory stream alert occurs, a node sends the following message:



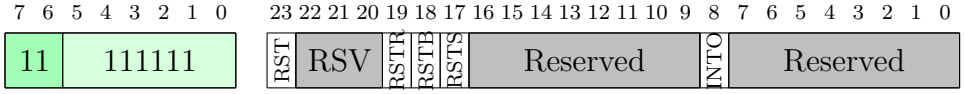
    - The MBus Address is set to the `Alert Address` specified by bits 23:16 in `R1110XX00`.
    - The top 8 bits of data are set to the `Alert Register to Write` specified by bits 23:16 in `R1110XX01`.
    - The next 8 bits are the `Alerting Stream Channel`, made up of this node's short prefix, followed by `01`, followed by the channel that generated the alert—this should be the same address used to write to this stream channel.
    - The `EN` bit reports the current state of the `EN` bit of the alerting channel when the alert was sent.

    - The WRP bit is set if the write that generated this alert reached the end of the stream buffer.
    - The DBLB bit is set if the write that generated this alert reached the halfway point of the stream buffer and double-buffering is active for this stream.
    - The OVFL bit is set if the write that generated this alert reached the end of the stream buffer and there was already a pending alert with the WRP bit set or if the write that generated this alert reached the halfway point of the stream buffer and double buffering is active for this stream and there was already a pending alert with the DBLB bit set.
  - It is valid for a node to put its own address in the Alert Address field, in which case, no bus transaction should be generated, but a local alert should still fire.

- {R10[15:0]}: Buffer Length-1.
  - Defines the length of the buffer.
- {R10[23]}: Enable (EN).
  - Controls whether this channel is enabled. If EN is 0, a node must not modify memory in response to a memory stream message.[6]
  - **Acknowledement/Interjection:** The behavior of a stream write when EN is 0 is undefined. Receivers are encouraged to interject and indicate receiver error, however they may exhibit any behavior, including ACK'ing the transaction and silently ignoring it.
- {R10[22]}: Wrap (WRP).
  - Defines node behavior when the end of the buffer is reached. If WRP is high, the Write Address Counter should reset to its original value. If WRP is low, the Write Address Counter value should be unchanged (it should thus be one past the end of the valid buffer) and EN should be set to 0.
- {R10[21]}: Double Buffer (DBLB).
  - Controls double-buffering mode. If double-buffering is active, the node should generate an alert halfway through the buffer in addition to at the end of the buffer. This mode is most useful when combined with WRP.
- {R11[15:0]}: Buffer Offset
- Specify an offset into the buffer. Users should be mindful that many systems may not support unaligned access.

[6]Implementation Tip: Any undefined MPQ registers are defined to be RZWI, that is a read from an undefined register will read as all 0's (and a write ignored). Upon a read, this will return 0 for EN as required. Nodes that do not implement a memory stream channel do not require any logic to handle any memory stream messages, simply leave the register undefined.

Reg #255: Action Register



<No Storage>

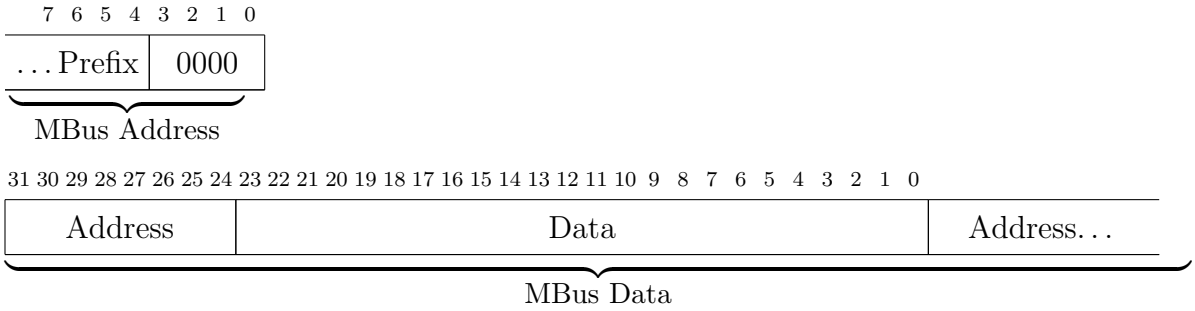
This register requests that an action be performed. It is an error to request more than one action in a single request. Actions are processed from MSB to LSB, that is, if more than one action is requested *only* the highest priority action is actually taken. A read from this register shall always return all `0`. A write of all `0` to this register shall perform no actions. If any bit written to this register is non-zero, writing this register **must** be the last operation in the transaction. The behavior of anything after a register action request in the same transaction is undefined.

- {`R[23]`}: `Reset (RST).`
  - Reset the entire node. The exact result of a reset is implementation-defined, however this request should be the most aggressive form of reset available.
- {`R[19]`}: `Reset MPQ Registers (RSTR).`
  - Reset MPQ configuration registers that affect register protocol behavior to their default value.
  - Resets #223–247.
- {`R[18]`}: `Reset Bulk Registers (RSTB).`
  - Reset MPQ configuration registers that affect memory bulk transfers to their default value.
  - Resets #242.
- {`R[17]`}: `Reset Stream Registers (RSTS).`
  - Reset MPQ configuration registers that affect memory stream transfers to their default value.
  - Resets #224–239.
- {`R[8]`}: `Interrupt Owner (INTO).`
  - Asserts the Layer Owner Interrupt.

Register Commands

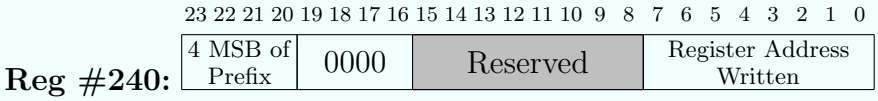
The MPQ register space is an 8 bit addressable array of 24 bit wide registers. Any undefined bits are treated as RZWI (Read as Zero, Write Ignored). This section expresses the on-wire interface to registers.

Register Write



Bits 0-23 of the MBus data field are written to the register addressed by bits 24-31. The write occurs immediately, as soon as the layer controller receives the message. Multiple registers may be written in a single MBus transaction by sending multiple data packets. Each 32 bit chunk of data is treated as if it were an independent transaction.

Record and interrupt (if enabled) after each register write:

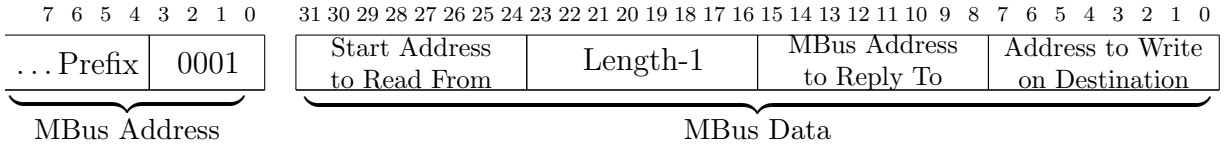


Overflow: If more data is received after writing the last register (`0xff`), the destination address wraps and registers continue to be written, beginning at address `0x00`.

Unaligned Access: The behavior of a message ending on a non-word boundary is undefined.

Interjection Semantics: Each command is applied immediately when it is received. A four-command message would have a $4 \times 32 = 128$ bit data payload. If 63 bits are received prior to interjection, only the first command was applied. If 64 bits are received, the first two commands are applied.

Register Read



Bits 24-31 specify the address of the register to be read. Bits 16-23 may be used to request that multiple registers are sent. This field is a count of the number of values to be sent less one, that is a value of 0 requests 1 register is read and a value of 255 requests that all 256 registers are sent. Bits 8-15 are the MBus address the reply is sent to. Bits 0-7 specify the first address field of the Register Write response.

The response message is sent to the MBus address specified in bits 8-15 of the request and its data field is formatted exactly as the Register Write command: 8 bit address + 24 bit

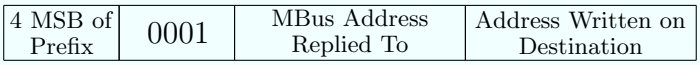
data. For reads of more than one register, the address field in the response is incremented by 1 for each register.

The response always sends the requested length. If a request for register #256 would have been made, the request wraps and begins from register #0. The destination register address wraps similarly.

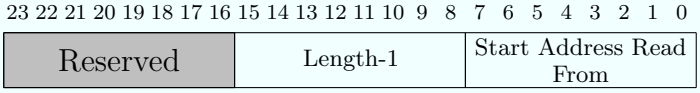
Note: The MBus address to reply to **must** be copied exactly. The FU_ID is not required to be Register Write. For example, if the FU_ID is Memory Stream Write, the effect is dumping the current register state to memory on the target addresses.

Record command information. Generate the whole response message. Interrupt (if enabled) after sending complete response.

Reg #240:



Reg #241:

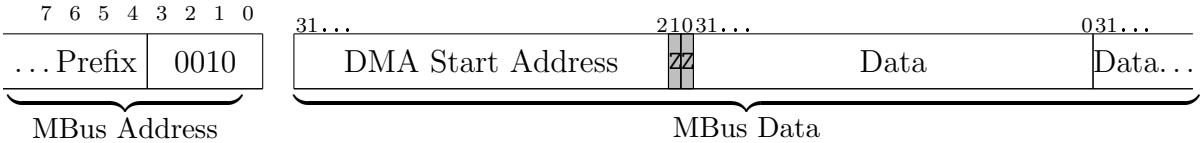


Interjection Semantics: If the reply is interjected, the transaction is aborted and is **not** retried.

Memory Commands

The MPQ memory space is a 32 bit addressable array of 32 bit words of memory. Any undefined accesses are treated as RZWI (Read as Zero, Write Ignored). MPQ provides two types of memory commands: bulk and stream. A bulk memory transaction is a wholly self-contained event designed for DMA of large blocks of memory. Memory streams pre-configure the stream information in MPQ registers on the destination node and omit it from subsequent transactions, relying on the receiver to maintain and increment a destination pointer. Streams are useful for applications such as continuous sampling, where multiple, short messages are generated.

Memory Bulk Write

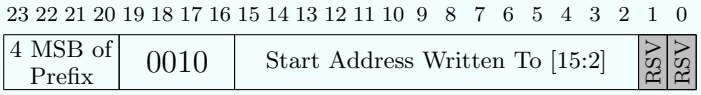


The first word received is the address in memory to begin writing to. The bottom two bits of the address field are reserved and **must** be transmitted as 0. Subsequent words are treated as data. The first word of data is written to the specified address. The next word of data is

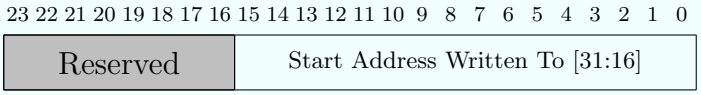
written to address+4 (the next word in memory) and so on. There is no limit on the length of this message.

Record command information. When the write completes, generate an interrupt (if enabled). If an error occurs part way through, the Reg #250 should indicate the number of words actually written.

Reg #240:

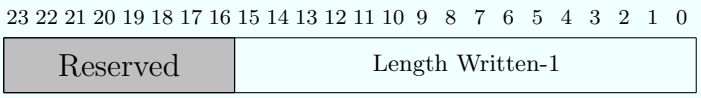


Reg #241:



Reg #242, #243 not written.

Reg #244:



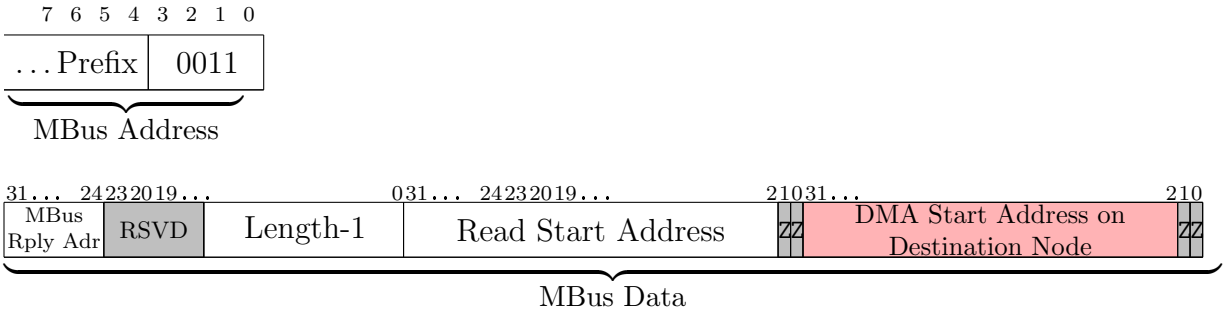
Implementation Note: These registers may be updated while the command is running, so long as they have the correct value once it completes. They are a good place to store the pointer and counter while the command is active instead of instantiating dedicated counter and address registers.

Unaligned Access: The behavior of a message ending on a non-word boundary is undefined.

Overflow: If more data is received after writing the last address in memory (`0xfffffffc`), the destination address wraps and data continues to be written, beginning at address `0x00000000`.

Interjection Semantics: Each word of data is written to memory immediately when it is received. A four-word message would have a $(1 + 4) \times 32 = 160$ bit data payload. If 95 bits are received prior to interjection, only the first word was written to memory. If 96 bits are received, the first two words were written to memory.

Memory Read

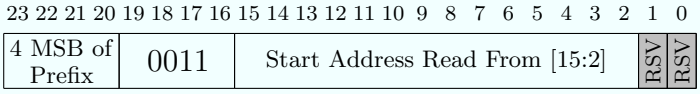


The first word indicates the MBus address to reply to and the length of the requested read in words less one. A length of 0 will reply with 1 word of data. The second word is the address in memory to read from. The bottom two bits of the address field are reserved and **must** be transmitted as 0.

The third word is **optional**. If the third word is present, it is prepended to the reply (generating a message with a Memory Bulk Write formatted payload). If the third word is omitted, data is immediately placed on the bus (generating a message with a Memory Stream Write formatted payload).

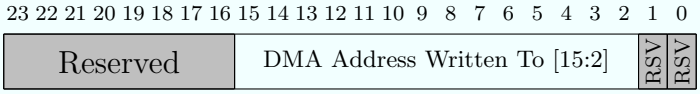
Record command information. Generate the response and send it. When the response completes, trigger an interrupt (if enabled). If an error occurs part way through, the Reg #250 should indicate the number of words actually written. If the optional third word is omitted, Reg #251–252 are undefined.

Reg #240:

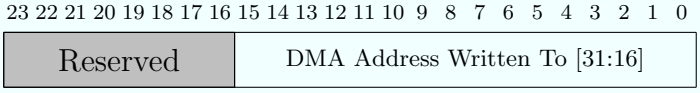


Reg #241:

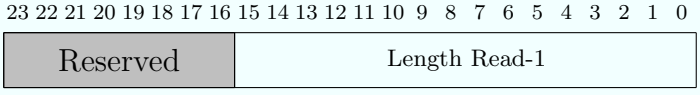
Reg #242:



Reg #243:



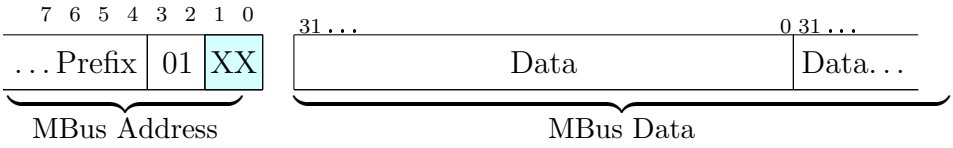
Reg #244:



Overflow: If the starting address field and subsequent length exceed the memory space, that is a request for address `0x100000000` would have been made during the response, the layer controller wraps and continues sending from address `0x00000000`.

Interjection Semantics: If the reply is interjected, the transaction is aborted and is **not** retried.

Memory Stream Write

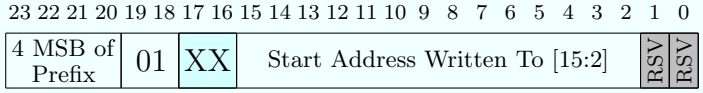


MPQ nodes have up to four streaming memory channels. Each channel is controlled by configuration registers (Reg #224–239: Memory Stream Configuration). The destination of a memory stream write is specified by a combination of channel selection—the last two bits of the FU_ID—and the pre-arranged configuration.

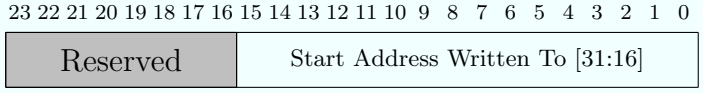
The message payload is all data. The destination address is automatically incremented every time a word is written. There is no limit on the length of this message.

Record command information. When the write completes, generate an interrupt (if enabled). If an error occurs part way through, the Reg #250 should indicate the number of words actually written.

Reg #240:

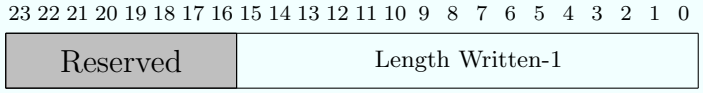


Reg #241:



Reg #242, #243 not written.

Reg #244:



Unaligned Access: The behavior of a message ending on a non-word boundary is undefined.

Overflow: If more data is received after writing the last address in memory (`0xfffffffc`), the destination address wraps and data continues to be written, beginning at address `0x00000000`.

Interjection Semantics: Each word of data is written to memory immediately when it is received. A four-word message would have a $(1 + 4) \times 32 = 160$ bit data payload. If 95 bits are received prior to interjection, only the first word was written to memory. If 96 bits are received, the first two words were written to memory.

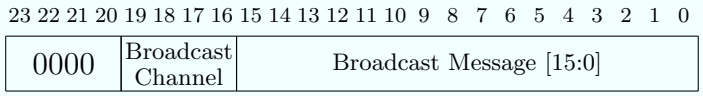
Broadcast Snooping

Broadcast Channel 0: Node Discovery and Enumeration and Broadcast Channel 1: `Layer` Power are built into the MBus protocol which runs below MPQ. However, it is possible that a MPQ node may wish to snoop broadcast traffic (in particular, Query/Enumerate Response).

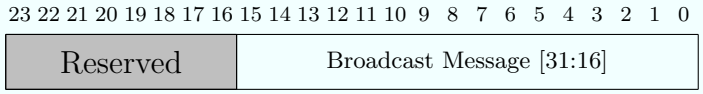
If broadcast snooping is active, whenever a broadcast message is received:

Record command information. Interrupt (if enabled) after message completes.

Reg #240:



Reg #241:



Undefined Commands

If command with an unknown FU_ID is received, MPQ behavior is completely undefined.

MPQ in Practice

These interface definitions enable general purpose any-to-any MMIO-like transactions. One of the key capabilities worth emphasizing again is that any third device can initiate such operations. To illustrate this, the following example[7] shows how the MPQ interface would issue a bulk memory (DMA) operation between two remote nodes:

```
void mbus_copy_mem_from_remote_to_any_bulk (
    uint8_t   source_prefix,
    uint32_t* source_memory_address,
    uint8_t   destination_prefix,
    uint32_t* destination_memory_address,
    uint32_t  length_in_words_minus_one
    ) {
  uint32_t payload[3] = {
    ( ((uint32_t) destination_prefix) << 28 )
      | (MPQ_MEM_BULK_WRITE << 24) | (length_in_words_minus_one & 0xFFFFF),
    (uint32_t) source_memory_address,
    (uint32_t) destination_memory_address,
  };
  mbus_write_message(((source_prefix << 4 ) | MPQ_MEM_READ), payload, 3);
}
```

[7] From https://github.com/lab11/M-ulator/blob/master/platforms/m3/pre_v20e/software/libs/mbus.c

4.4 Summary

This chapter assigned semantic meaning to interconnect messages. In contrast to traditional embedded interconnects, which describe only how to read or write arbitrary bits and bytes, MPQ defines standard register and memory abstractions and how to read and write them. This extends the reach of system composition and creates a "namespaced MMIO" abstraction. In this way, composable design concepts can flow freely across chip boundaries, which softens the previously hard delineation between system and system-on-chip. This chapter completes the design of the modular interconnect for Smart Dust systems. The next chapter puts these ideas to test, starting with theoretical analyses and progressing to the implementation of several real-world millimeter-scale systems.

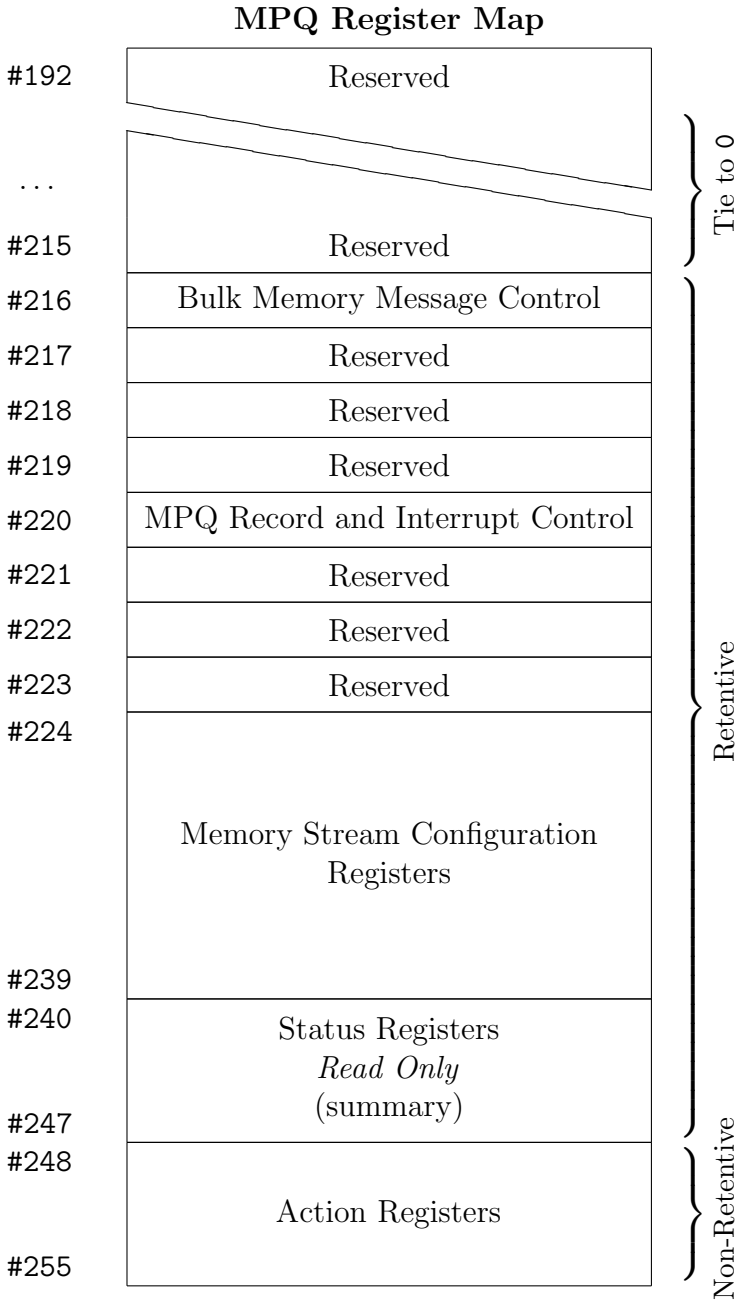


Figure 4.1: **MPQ register map.** MPQ defines a portion of the register space for its own operation. This standard interface allows portable software libraries and remote DMA operation.

Chapter 5

The M3 Implementation

This chapter describes one specific instantiation of MBus and MPQ, as built for the Michigan Micro Mote (M3) project. This implementation is chosen as a case study since it leverages all of the power optimization faculties of MBus. It is further an opportunity to understand how MBus is designed to integrate into both the individual pieces of a system as well as system operation at large.

5.1 Overview

Figure 5.1 shows the complete Verilog design. For non-power-conscious designs, the blocks in green and power control signals can be omitted. To support self and system power-gating, the additional Sleep Controller, Wire Controller, and optional Interrupt Controller are added. The Layer Controller implements MPQ and serves as the interface to the rest of the node. While most nodes will likely have a local oscillator, it is possible to interface with the Bus Controller using only edges harvested from the bus clock.

5.2 RTL Design

The M3 MBus implementation defines two major components: a *Bus Controller* and a *Layer Controller*. The Bus Controller understands the MBus protocol and presents a simple word-wide interface to higher layers. The generic Layer Controller provides a register file and a memory interface, sufficient for most devices.

In addition, the M3 MBus implementation requires some support blocks: a *Sleep Controller*, a *Wire Controller*, and an *Interrupt Controller*. They are hand-optimized (during layout), always-on components designed for minimal power draw.

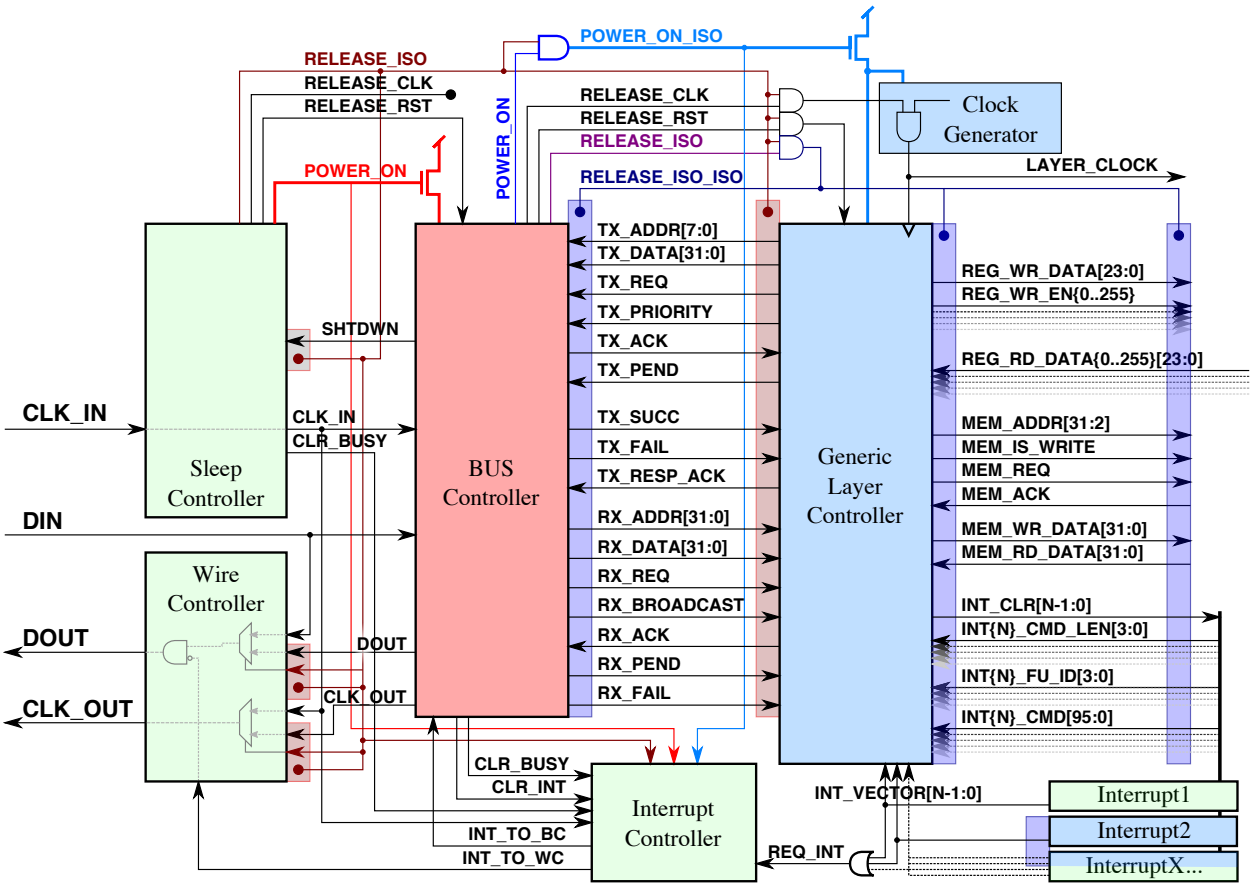


Figure 5.1: **MBus implementation.** MBus is implemented as a series of composable Verilog modules. The module coloring represents the three hierarchical power domains: green modules (Sleep, Wire, and Interrupt Controllers) are always powered on, red modules (Bus Controller) are powered during MBus transactions, and blue modules (Layer Controller, Local Clock) are powered only when the node is active. Critically, MBus itself requires no local oscillator. The generic Layer Controller implements and provides a simple, consistent register/memory interface for a node. The isolate (`ISO`) signals ensure that floating signals from power-gated blocks remain at stable defaults. Systems that do not perform power-gating omit these isolation gates and all of the green blocks.

Sleep Controller

The job of the Sleep Controller is to wake the Bus Controller when a message on the bus begins. The Sleep Controller is also responsible for powering down the Bus Controller when requested, using edges from MBus to do so. For layers capable of waking on events other than MBus transactions, the Sleep Controller may be extended or need to coordinate with other Sleep Controllers.

Power Signals

The Sleep Controller uses both edges of the CLK_IN signal to generate the power signals. The rising edge that resolves arbitration is used to release the power-gating. The falling edge (Priority Drive) is unused and reserved for implementations that require a RELEASE_CLK signal. The next rising edge (Priority Latch) drives RELEASE_RST and the falling edge (Begin Transmission) drives RELEASE_ISO. The next rising edge latches the Reserved bit from the bus, which is currently unused and results in an empty cycle.

MBus Signals

The Sleep Controller samples and passes through the MBus CLK_IN signal. The Sleep Controller uses this line to clock its internal state machine. The Sleep Controller and the Bus Controller can be safely considered synchronous modules with respect to one another as they both rely on the MBus CLK_IN signal to provide a clock.

Module Signals

The Sleep Controller has only one non-reset input signal: SHTDWN. This signal is asserted by the Bus Controller to request that it be put to sleep. This signal is sampled synchronously with the MBus clock. The SHTDWN signal may only be asserted during the falling edge Drive Control Bit 1.

The Sleep Controller is also responsible for generating a CLR_BUSY output signal when it powers off the Bus Controller. This signal is normally generated by the Bus Controller at the end of MBus transactions and is used to indicated to the Interrupt Controller that the bus is no longer busy.

Wire Controller

The Wire Controller is responsible for physically driving the MBus wires. This separation is necessary to ensure that the data and clock lines are forwarded even when the Bus Controller is power-gated. The Wire Controller is a very simple module, conceptually it is little more than a mux that either forwards the values coming in from the bus or values coming from the Bus Controller.

Changing the Wire Controller muxes requires extreme care to ensure that unintentional glitches are not introduced. In practice, this means only changing the clock mux signal when CLK_IN is already high. The data mux is more complicated. During Idle, the data line is high, any glitches while pulling it low (so long as it ultimately is held low) will be significantly shorter than the mediator wakeup and therefore ignored. During transmission, the state of the data line is not always known when control transitions are required. MBus ignores the data line while the clock is low. Any potentially glitch-inducing changes to the data mux must occur on the falling edge of clock to avoid uncertainty.

The Wire Controller also facilitates external interrupts as a wakeup source via a glitch inducer, as discussed in Section 3.3. The goal is to incite a bus arbitration cycle but not participate in it. Usually, this procedure results in no winner of arbitration, and the bus will reset. However, the glitch inducer logic must also correctly handle the case where another node was performing genuine arbitration at the same time, and a real message takes place.

Interrupt Controller

The Interrupt Controller is responsible for directing interrupt events to the appropriate modules at the appropriate times. The destination of an interrupt event depends on the current power state of the Bus Controller and Layer Controller.

If the Layer Controller is powered on, the Interrupt Controller blocks the interrupt signal as the Layer Controller will handle the interrupt and generate any appropriate messages. If the Layer Controller is powered off, however, the Bus Controller must wake the Layer Controller. To do this, the Bus Controller must harvest edges from MBus. To generate the power signals, the Bus Controller must observe a transaction while the `INT_TO_BC` signal is asserted. To generate these edges, the Interrupt Controller must induce a glitch.

A Subtle Detail: Extreme care must be taken when this glitch is induced. In particular it is important to ensure that MBus is idle so that a real glitch is not created. The simple approach would be to ask the Bus Controller to simply export a "bus busy" signal, asserted while a transmission is active. Such a signal is not sufficient, however. If a Bus Controller is not powered on, it is incapable of asserting busy. If two nodes have interrupts near each other in time, the first node will induce a glitch unobserved by the second node. If an interrupt occurs on the second node between end of the mediator wakeup and resultant falling edge of `CLK_IN` and the rising edge for arbitration, the second node will unintentionally (and worse unknowingly) win arbitration. As the node does not know that it has won arbitration, it will never end the transmission and the bus will hang until the mediator's watchdog expires.

To avert this issue, the Interrupt Controller keeps an internal sense of "bus busy". The Interrupt Controller latches the bus as busy whenever `CLK_IN` goes low. The internal busy signal is then cleared by an explicit signal from the Bus Controller at the end of MBus transactions. As a special case, the Sleep Controller must clear the busy status when it powers the bus controller down as the Bus Controller will be powered off at the end of the transaction and incapable of generating the signal. This clear busy signal is internally treated as the last step in the power-down sequence.

Bus Controller

The Bus Controller is responsible for handling all possible bus events, including generating acknowledgments if it is the target of a transaction. The Bus Controller is only obligated to hold one word of a transaction, devices wishing to receive messages longer than one word in length must have a local FIFO to store partial messages in their Layer Controller.

Power Signals

The Bus Controller provides the same power control signals to the Layer Controller that the Sleep Controller provided to the Bus Controller. For basic layers, the Bus Controller is responsible for waking and sleeping the Layer Controller.

The Bus Controller should only wake the Layer Controller if there is a message of *non-zero* length being transmitted to the layer's address. In practice this means the Bus Controller should not begin waking the Layer Controller until it has received the *third* data bit. Layer Controllers are not awoken for broadcast messages as all currently defined broadcast messages can be handled by the Bus Controller alone.

Once the Bus Controller elects to begin waking the Layer Controller, it **must** wake the Layer Controller completely. Even if another data bit is never sent (due to an interjection) after the Bus Controller elects to wake the Layer Controller, there are still the Begin Interrupt, Control Bit 0, Control Bit 1, and Begin Idle edges to use to wake the Layer Controller (which means it is sufficient to rely on solely positive edges to drive the wakeup state machine).

The Bus Controller will indicate an erroneous wake-up by asserting `RX_FAIL` signal. The Layer Controller must acknowledge (`RX_ACK`) the erroneous transmission before the Bus Controller is capable of receiving another message. This enables the Layer Controller to take action to put itself back to sleep after a spurious wakeup.

MBus Signals

While the Bus Controller is not directly connected to the external pads for all MBus signals, it does use all MBus inputs:

INPUT	`CLK_IN`	Bus Clock In
OUTPUT	`CLK_OUT`	Bus Clock Out
INPUT	`DIN`	Bus Data In
OUTPUT	`DOUT`	Bus Data Out

Module Signals

INPUT	`RESET`	Global system reset
OUTPUT	`SHTDWN`	Request power-gating
INPUT	`TX_ADDR[7:0]`	Address to transmit
INPUT	`TX_DATA[31:0]`	Data to transmit
INPUT	`TX_REQ`	Request to transmit
INPUT	`TX_PRIORITY`	Is high priority message?
OUTPUT	`TX_ACK`	Acknowledge request to transmit
INPUT	`TX_PEND`	More data pending

OUTPUT	TX_SUCC	Transmit successful
OUTPUT	TX_FAIL	Transmit failed
INPUT	TX_RESP_ACK	Acknowledge TX successful / fail
OUTPUT	RX_ADDR[31:0]	Destination address of RX'd packet
OUTPUT	RX_DATA[31:0]	Data received
OUTPUT	RX_REQ	Data is ready
OUTPUT	RX_BROADCAST	RX'd message was a broadcast message
INPUT	RX_ACK	RX_DATA has been saved
OUTPUT	RX_PEND	More data is coming
OUTPUT	RX_FAIL	Abort current RX

The Bus Controller provides a single-word interface to MBus to higher level modules. To ensure that the Bus Controller can send a timely ACK for a received message, modules are obligated to be able to (eventually) receive a minimum of one word. That is, if the RX_REQ line is raised, the module *must* eventually receive that word by signaling RX_ACK. If another message is received while RX_REQ is still high, the Bus Controller will Interrupt the bus indicating "01" (∼EoM, TX/RX Error). This serves to (i) save bus bandwidth by canceling a message that will not be received, and (ii) indicate to the transmitting layer that there is still a pending message (or message part) in the receiving layer.

If the TX_PEND signal is asserted but a TX_REQ for a new word is not sent by the time the first word has been sent, the Bus Controller will Interrupt the bus indicating "01" (∼EoM, TX/RX Error). The Bus Controller will also assert TX_FAIL, at which point the transaction must be aborted.

The RX_PEND signal requires special attention. When the RX_REQ signal rises, if RX_PEND is also high, it indicates that there is more data to follow. If a module asserts RX_ACK in response it is obligating itself to receive at least one more word after the current word.[1] If a module cannot receive another word beyond the word it is currently latching, it should ignore RX_REQ. When the next word is received by the Bus Controller, it will detect that RX_REQ is still high and abort the entire transaction, Interrupting to indicate RX Error. In practice this means a transmitting node may believe it has sent one more word than was actually received. As the entire transaction is NAK'd however, the only implications are for the software flow control estimation, which can compensate.

For simple nodes that only support single word transactions, this RX_PEND subtlety is important. Such a node should wire its final acknowledgment output signal something like

```
TX_ACK_out = (RX_PEND_in) ? 1'b0 : internal_ack_signal;
```

to ensure it does not attempt receipt of multi-word transactions. The TX_PEND signal can simply be tied low.

[1] This is logically a continuation of the original contract. In an idle state, a module has received zero words thus far and is obligated to be able to receive one more. If a module ACKs a word while RX_PEND is high, it is accepting the current word *and* committing to receive the next pending word.

Parameters

The Bus Controller requires relatively little configuration. The only available parameter is the address(es) that this bus instance should respond to:

`ADDRESS`	Address(es) to receive and acknowledge
`ADDRESS_MASK`	Which bits of `ADDRESS` are significant

Generic Layer Controller

The Layer Controller is responsible for communicating with the Bus Controller and facilitating multi-word transactions. It federates access to the individual components on a MBus member node. The generic Layer Controller is designed such that unused modules (e.g. memory access and control) can be synthesized out if they are unused.

Power Signals

The generic Layer Controller can only be woken by the Bus Controller. Layers capable of generating an interrupt while M3 is in sleep mode (e.g. an alarm) must wire into the glitch inducer via the Interrupt Controller to wake the layer and the whole M3 system. The generic Layer Controller does not output any power signals.

There is no explicit shutdown signal from the Layer Controller to the Bus Controller. Rather, the Layer Controller issues a broadcast message announcing to the bus its intention to shut down. The Bus Controller will recognize the special broadcast message and shut down the Layer Controller once it is sent. The actual shutdown does not begin until the Bus Controller successfully transmits the `EoM` bit at the end of the transaction.

Module Signals (Bus Side)

All Layer Controllers have a common set of signals to interface with the Bus Controller, as shown in Figure 5.1. Details of these signals are presented in the Bus Controller section in the Module Signals description.

Module Signals (Interface Side)

The Layer Controller expects to communicate with two types of hardware: a register file and a basic memory. One or both of these may be instantiated. They are treated independently by the Layer Controller.

Register File: The register file is for small, simple configuration, status, or action bits. It presents an 8 bit address space with 24 bit wide data. Writes are pulse-triggered, with a unique write control line for each of the registers. Data to read must always be valid, with no indication that data is being read. Registers may be smaller than 24 bits. Unused bits for writing should be left disconnected. Unused bits for reading must be tied low.

OUTPUT	REG_WR_EN{0..255}	Register write enable lines
OUTPUT	REG_WR_DATA[23:0]	Register data
INPUT	REG_RD_DATA{0..255}[23:0]	Wires from registers

Memory: The memory interface is for larger amounts of data (images, audio, packets, etc). Memory is a 32 bit address space with 32 bit data. The memory space does not alias the register file address space.

The memory interface is a simple two-wire handshake that indicates a request when signals are valid and an acknowledgment when the data has been latched. Memory must be fast enough that it can support streaming reads / writes at the line speed of MBus (see Relative Clock Frequencies for detail).

OUTPUT	MEM_ADDR[31:2]	Memory address
OUTPUT	MEM_IS_WRITE	Read/Write selection
OUTPUT	MEM_REQ	Request is ready
INPUT	MEM_ACK	Response is ready
OUTPUT	MEM_WR_DATA[31:0]	Data to write
INPUT	MEM_RD_DATA[31:0]	Data that was read

Module Signals (Interrupt Interface)

To generate messages internally, the local node interrupts the Layer Controller. Interrupt priority is fixed and ranked by the interrupt index. Interrupts are non-interruptible and are serviced completely before handling the next or new interrupt. If an interrupt and a message from the bus arrive at the same time, the interrupt takes priority.

Conceptually, a local interrupt "fakes" the receipt of a message from the bus, thus the data format is the exact same as bus commands. To generate a memory *write* command for example, the interrupt payload is a memory *read* command, because the response to a memory read request is to generate a memory write. The CMD_LEN specifies the length in words of the payload that is valid. A length of zero indicates that no command should be executed (the FU_ID and CMD fields are ignored) and can be cleared immediately. This is useful for generating wakeup requests with no immediate command to execute.

INPUT	INT_VECTOR[N-1:0]	Interrupt requests
OUTPUT	INT_CLR[N-1:0]	Clear interrupts
INPUT	INT{N}_CMD_LEN[1:0]	Word length of payload
INPUT	INT{N}_FU_ID[3:0]	Command the Layer Controller "receives"
INPUT	INT{N}_CMD[95:0]	Command payload

Module Interdependencies

In addition to the module signals, the following additional requirements must be considered for the blocks to operate correctly.

Clock Domains

The Bus Controller and Layer Controller are on separate clock domains. Care must be taken with the `REQ` and `ACK` two-wire handshake to ensure that signals are all double-latched. Double latches should not be required for the other signals are they are only sampled after the `REQ` or `ACK` line is stable.

The generic Layer Controller does not perform double-latching with any of the signals attached to the register file or memory. These blocks are expected to run on the same clock as the Layer Controller. Designs that violate this assumption must make modifications to ensure signal stability.

Relative Clock Frequencies

Nominally, the Layer Controller would only require activity once every 32 MBus clock cycles. In practice, however, there is a double-latched, bidirectional handshake between the Bus Controller and Layer Controller. At a minimum then, the Layer Controller must be at least 1/8 the speed of the bus. This timing constraint further extends to the memory if longer bus transactions are going to be supported. The generic Layer Controller is a write-through device, it does not buffer memory requests. Consider the memory latency in cycles M to be defined as the maximum possible number of cycles between the assertion of `MEM_REQ` before the `MEM_ACK` response is asserted, then the minimum clock speed of the Layer Controller and memory is $\frac{6+M}{32}$ of the bus speed. There is no upper bound on the layer speed relative to the bus. Layer Controller clock speeds are generally configurable.

5.3 Evaluation

In many regards the most significant evaluation of this work will be time. Will the availability of modularity accelerate the introduction of Smart Dust, and will the resulting Smart Dust actually use the interface principles and designs outlined here? Only time can tell.

It is, however, possible today to evaluate whether MBus achieves its design goals. Towards that end, this section first considers the efficiency and baseline efficacy of the proposed implementation. It next looks in depth at MBus, first via an evaluation of the protocol and then theoretical and empirical energy performance analysis. Then two representative systems from the Michigan Micro Mote (M3) ecosystem are examined to see how well real-world systems behave. Finally, there is some consideration of future expansion via scalability and interoperability of MBus.

Figures of Merit

Table 5.4 shows the cost in area for each of the MBus components (excluding I/O pads) when synthesized for an industrial 180 nm process, with comparisons to SPI, I^2C, and Lee's I^2C variant. MBus imposes an area cost penalty, but offsets this with its additional features.

Module	Verilog SLOC	Gates	Flip-Flops	Area in 180 nm
Bus Controller	947	1314	207	27,376 µm^2
Optional				
Sleep Controller	130	25	4	3,150 µm^2
Wire Controller	50	7	0	882 µm^2
Interrupt Controller	58	21	3	2,646 µm^2
Total	1185	1367	214	37,200 µm^{2§}
		Other Buses:		
SPI Master†	516	1004	229	37,068 µm^2
I^{2}C ‡	720	396	153	19,813 µm^2
Lee I^{2}C [17]	897	908	278	33,703 µm^2

§ Includes a small amount of additional integration overhead area
† SPI Master from OpenCores [68] synthesized for the same 180 nm process
‡ I^{2}C Master from OpenCores [69] synthesized for the same 180 nm process

Table 5.4: **Size of MBus components.** Non power-gated designs require only the Bus Controller. The MBus values are from the temperature sensor chip in Figure 5.5. To support its additional features and lower power, MBus incurs a modest increase in area.

Protocol Evaluation

Topology

Because MBus is a ring, as the number of nodes increases, so does the propagation delay around the ring. The MBus specification defines a maximum node-to-node delay of 10 ns, which is achieved by all of designs to date. Figure 5.2 explores how node count affects the bus clock and finds that a 14-node MBus system can run at up to 7.1 MHz. As a reference, I^{2}C clock speed ranges from 100 kHz (Standard) to 5 MHz (Ultra Fast) [51]. Some special-purpose SPI implementations reach speeds as high as 100 MHz, though most low-power microcontrollers have an upper limit of 16 MHz for the I/O clock [70, 71].

Overhead

In addition to transmitting data, MBus transactions require arbitration (3 cycles), addressing (8 or 32 cycles), interjection (5 cycles), and control (3 cycles), an overhead of 19 or 43 cycles depending on the addressing scheme. Figure 5.3 compares MBus overhead to other common buses and finds that MBus's length-independent overhead is more efficient after 9 byte payloads than length-dependent protocols, without incurring significantly greater overhead for shorter messages.

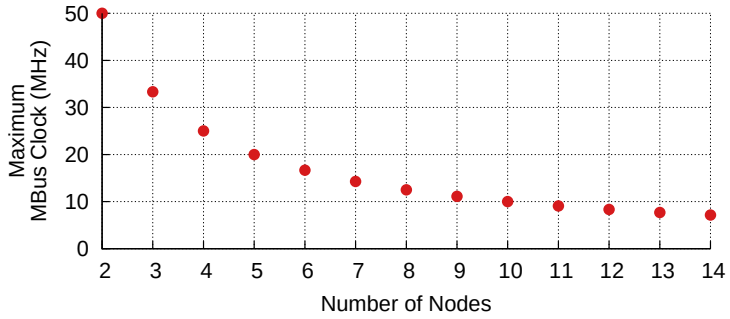


Figure 5.2: **Maximum frequency.** MBus peak clock frequency is inversely proportional to the number of nodes. MBus limits node-to-node propagation delay to 10 ns. For the maximum of 14 short-addressed nodes, MBus could support a 7.1 MHz bus clock.

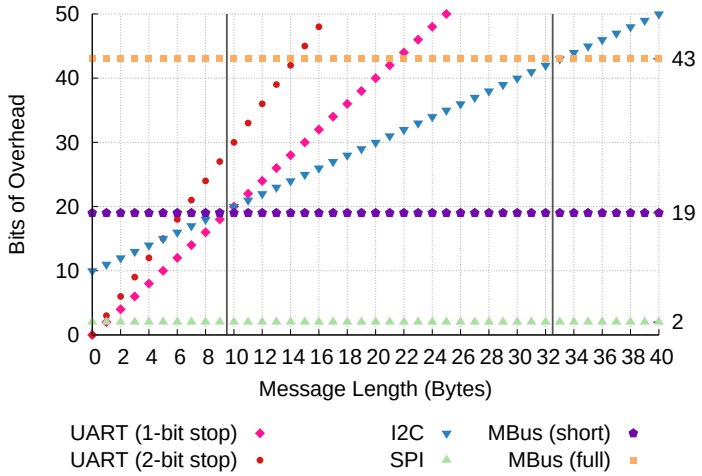


Figure 5.3: **Bus overhead.** MBus message overhead is independent of message length. MBus short-addressed messages become more efficient than 2-mark UART after 7 bytes and more efficient than I^2C and 1-mark UART after 9 bytes. MBus scales efficiently to messages such as 28.8 kB images (Section 5.3) or longer.

Power

Simulation

To capture a detailed estimate of the cost of running MBus, the implementation is run through Synopsis PrimeTime. The model is post-APR and uses standard wire models. To model I/O load, the simulation uses a conservative pad model, which estimates 2 pF per pad. The simulation estimates that MBus draws 5.6 pW per chip in idle and consumes 3.5 pJ/bit/chip while transmitting. From this, the energy estimate of a single MBus message is:

$$E_{message} = [3.5\,\text{pJ} * (\{19 \text{ or } 43\} + 8 * n_{bytes})] * n_{chips}$$

		Energy per bit
Member+Mediator Node	sending	27.5 pJ/bit
Member Node	receiving	22.7 pJ/bit
Member Node	forwarding	17.6 pJ/bit
Average		22.6 pJ/bit

Table 5.5: **Measured MBus power draw.** As supplies for MBus modules are not broken out, their power draw cannot be directly measured. Instead, this uses differential system states, which measures how much total system power draw changes when various MBus operations are active, to capture an estimate of the power draw for MBus. Forwarding nodes reduce switching activity by not clocking flops in their receive buffer. The mediator is integrated as a block in the processor and cannot be isolated.

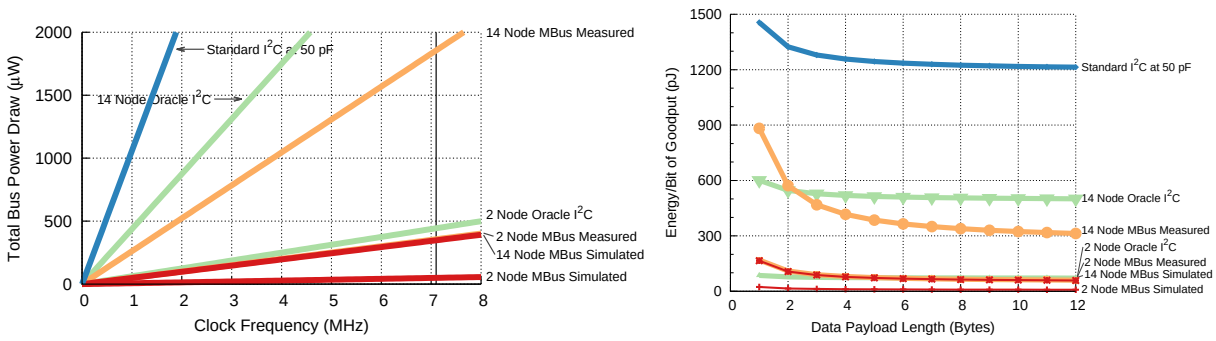
Measurement

Empirical power measurements are captured on a debug (non-stacked) version of the temperature sensor shown in Figure 5.5 and evaluated in Section 5.3. Although the power draw of MBus cannot be directly measured in the fabricated chips, it is possible to measure the draw of each chip in the system in different states. The mediator is integrated as a block on the processor chip and cannot be disentangled from the rest of that chip's power draw. To get stable measurements, the processor is placed in a continuous loop sending invalid commands to the sensor node, which ignores them. As only the processor can be configured to send continuous messages, this measurement technique can report only the combined mediator and transmit energy consumption. The results of these measurements are summarized in Table 5.5. The simulation number is only the energy consumed by MBus directly. The approximately 6.5× increase over simulation can mostly likely be attributed to overhead such as internal memory buses and other integrated components that could not be isolated.

There is not an obvious means to directly or indirectly measure the power draw of MBus when idle. The total idle power draw of the temperature system is 8 nW, three orders of magnitude above the expected static leakage of MBus (5.6 pW) and comparable with the idle system power of the prior state-of-the-art. From this, it is reasonable to conclude that MBus contributes negligible power to the idle state.

Comparison to I^{2}C

The biggest inefficiency in I^{2}C stems from overprovisioning. Since the total bus capacitance is unknown, a power-inefficient, smaller value resistor must be chosen to guarantee timing constraints are met. Instead, one might imagine an "Oracle I^{2}C", in which the exact bus capacitance is known and an ideally large resistor is selected. To further improve Oracle I^{2}C power performance, one can allow the rise time to take the entire half clock period (zero setup and hold time) and treat 80% V_{DD} as logical `1`. This Oracle I^{2}C can then be modeled using the same simulation parameters as MBus (1.2 V, 2 pF/pad, 0.25 pF/wire). Figure 5.4



(a) Total Power Draw (b) Energy of Goodput Bits

Figure 5.4: **Energy comparisons.** First, (a) compares the power draw of various bus configurations as clock frequency and node population increase. It finds that both simulated and measured MBus outperforms the simulated Oracle I^{2}C, which itself outperforms standard I^{2}C. Then, (b) examines MBus overhead by computing the energy per bit for each bit of goodput, actual data bits that amortize protocol overhead. The simulated MBus outperforms the simulated Oracle I^{2}C for all payload lengths. The measured MBus reveals that MBus efficiency suffers for short (1–2 byte) messages and that systems should attempt to coalesce messages if possible. In both figures, the measured values are based on empirical measurements of the 3-node temperature sensor.

compares the performance of MBus simulation, an extrapolation of measured MBus values, Oracle I^{2}C simulation, and standard I^{2}C. Both simulated and measured MBus outperform Oracle I^{2}C for all but the shortest ($1-2$ byte) messages.

Microbenchmarks

These microbenchmarks examine two systems from the Michigan Micro Mote library of Smart Dust systems that are representative of typical embedded workloads. These demonstrate the importance of multi-master capability, efficient handling of large messages, and power-conscious design.

Sense and Send

Figure 5.5 shows a temperature sensor. This system is an archetypal "sense and send" design. The environment is periodically sampled and the reading is communicated from the sensor. In this system, the processor node periodically requests a temperature reading from the sensor node. In the request (4 bytes), the sensor node can be instructed to send the response (8 bytes) directly to the radio node, which transmits the message. These requests are infrequent (every 15 s) and short in duration, leading to a bus utilization of only 0.0022% at 400 kHz.

While transmitting the message directly from the sensor to the radio does reduce total bus utilization by 40%, that resource is not contested in this case. Energy, however, is always a

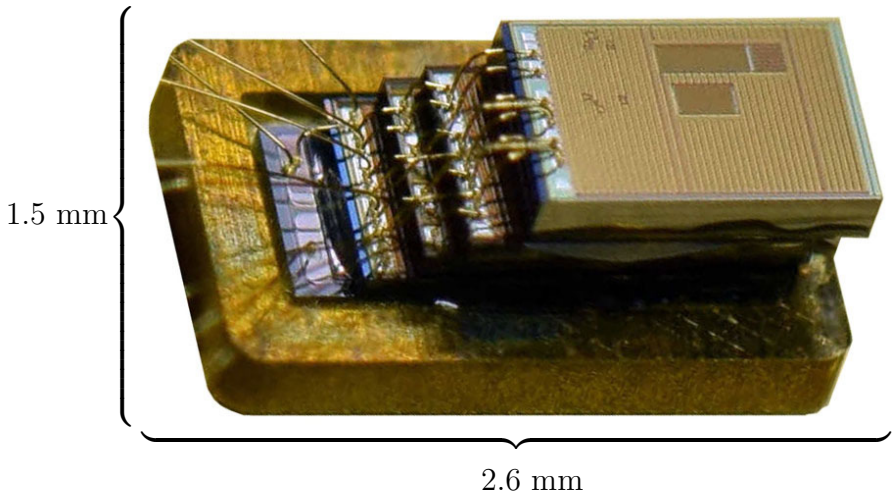


Figure 5.5: **Temperature sensing system.** A system designed as part of the Michigan Micro Mote project consisting of a 2 µAH battery, a 900 MHz near-field radio, an ARM Cortex M0 processor, and an ultra-low power temperature sensor, interconnected using MBus.

concern. In this three-chip stack there is one sender, one receiver, and one forwarder; sending an 8 byte message requires

$$(64 \text{ bits} + 19 \text{ bits}) \times (27.45 \frac{\text{pJ/bit}}{\text{TX}} + 22.71 \frac{\text{pJ/bit}}{\text{RX}} + 17.55 \frac{\text{pJ/bit}}{\text{FWD}})$$

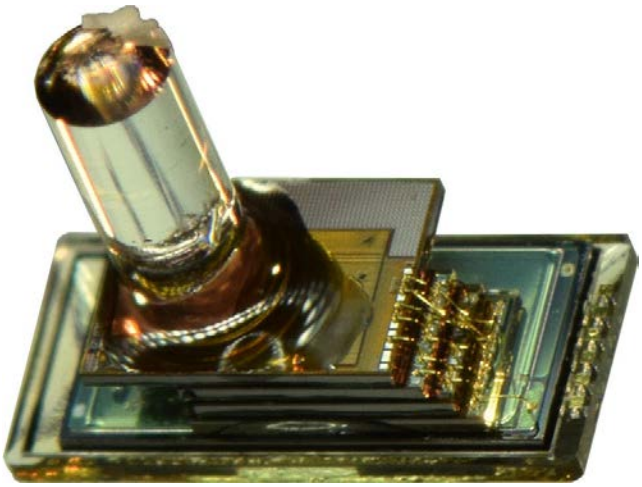
= 5.6 nJ; sending it twice would require 11.2 nJ. Further energy savings come from not powering on the processor. The processor uses approximately 20 $^{\text{pJ}}/_{\text{cycle}}$ and requires about 50 cycles to handle an interrupt and copy an 8 byte message to be sent again, using

$$50 \text{ cycles} \times 20 \frac{\text{pJ}}{\text{cycle}}$$

= 1 nJ. To collect an empirical estimate of the energy cost of an entire sense and send sequence, the system is programmed to sense and send continuously. Using the average power draw and sample rate during this loop, each sense and send event requires about 100 nJ of energy. By supporting any-to-any communication, MBus reduces the energy consumption of each sense and send event by 6.6 nJ (roughly 7%). Using the crude battery capacity of approximation of 2 µAh×3.8 V = 27.4 mJ, for a 15 s sample interval this increases node lifetime by 71 hours, from around 44.5 to around 47.5 days.

Monitor and Alert

For a second system, the motion-activated camera seen in Figure 5.6, exemplifies a typical monitor, filter, and alert system, and it demonstrates the need and efficacy of MBus's power faculties and efficient handling of large messages. During ultra-low power motion detection,



(a) Integrated System

(b) Captured Image

Figure 5.6: **Motion detection and imaging system.** (a) An imager made of a 900 MHz near-field radio, a 5 µAH battery, an ARM Cortex M0, and a 160 × 160 pixel, 9-bit grayscale imager with ultra-low power motion detection all connected using MBus. (b) A full-resolution (28.8 kB) image that was transferred by MBus.

the imager power-gates nearly all of its logic to minimize leakage. When motion is detected, the motion detector simply needs to assert one wire for MBus to wake the chip. By decoupling power management, the motion detector may act as a simple, standalone circuit or as a trigger to enter the more power-hungry image capture state whenever motion is detected.

The imager itself is a 160 × 160 pixel CMOS camera with 9-bit single-channel (grayscale) resolution. A full resolution image is 28.8 kB. The implemented MBus clock is run-time tunable from 10 kHz to up to 6.67 MHz (default is 400 kHz). Transferred as a single message, a full resolution image could take from 4.2 ms (238 fps) to 2.9 s (0.3 fps) depending on clock speed. Like most CMOS imagers, however, the camera reads pixels out one row at a time. To better cooperate with other possible bus users, the camera sends each row as a separate message, with small delays in-between while the next row is read out. Recall the correlation from Figure 5.3 between MBus message length and efficiency. By sending 160 180-byte messages instead of one 28.8 kB message, the image transmission incurs an additional 3,021 bits or 1.31% of overhead. By comparison, I^2C would incur 28,810 bits (12.5%) of overhead transmitting the whole image and 30,400 bits of overhead (13.2%) if sent row-by-row. MBus's message-oriented acknowledgment protocol results in a $90 - 99\%$ reduction in overhead compared to a byte-oriented approach.

Many-Node Systems

Both of these microbenchmarks are fundamentally one-sensor systems. One possible concern of the MBus design is how well it will scale to a greater number of connected nodes and what

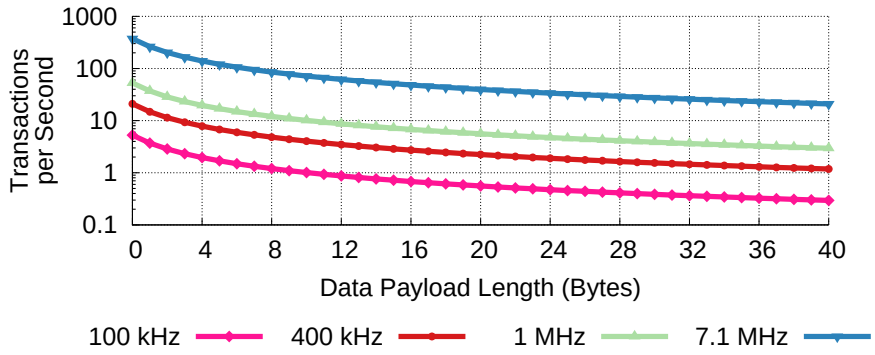


Figure 5.7: **Saturating transaction rate.** As a shared medium, MBus can only support a finite number of transactions across all member nodes. The peak transaction rate depends on the transaction size and bus clock speed.

the impact on lower priority nodes will be. A large and shared bus, ring, or interconnect topology, is not uncommon in an embedded design. Most microcontrollers have only one or two I^2C and/or SPI interfaces. I^2C also has a fixed priority scheme, based on the target address instead of the physical location of the sender. SPI is more flexible, allowing the central controller to select a priority scheme dynamically, at the cost of requiring a central controller to do so. The more important metric is not node count, rather it is the desired transaction rate – barring protocol overhead, for any bus two nodes sending messages at 1 Hz yields the same utilization as one node sending at 2 Hz. Figure 5.7 considers possible rates of MBus transactions as a function of message length. For brief periods of burst transactions that exceed the saturation rate, MBus provides both physical and logical mechanisms to enable system designers to federate bus access.

Interoperability

A key design goal was to facilitate interoperability independent of the technology used to fabricate MBus without requiring any tuning or tweaking. As a series of singled-ended connections—totem-pole FETs driving gates—the MBus design is well-suited to meet this constraint. As evidence of this claim, the systems demonstrated here integrate chips from 65, 130, and 180 nm processes from two different fabs. The chips also interoperate with outside hardware, and in testing have been verified with debug interfaces from an NI board [72] and a Microsemi IGLOO nano FPGA [73]. Newer MBus chips add built-in level converters for I/O pins, however all of the chips tested in this work operated at 1.2 V. In well over 1,000 hours of system testing, we are yet to encounter any MBus-related issues.

Bitbanging MBus

No existing commercial microcontroller includes an MBus frontend. To investigate MBus viability on existing microcontrollers without a dedicated MBus interface requires bitbanging implementation. MBus requirements are modest. A generic C implementation[2] requires only 414 SLoC and requires only four GPIO pins (two must have edge-triggered interrupt support). To estimate the overhead of bitbanging MBus, we target an MSP430 [70] using `msp430-gcc-4.6.3` [74] and find that the worst case path is 20 instructions (65 cycles including interrupt entry and exit) to drive an output in response to an edge. With an 8 MHz system clock speed, the MSP430 can support up to a 120 kHz MBus clock. For a comparison, compiling[3] Wikipedia's I^{2}C bitbang implementation results in similar overhead, with a longest path of 21 instructions [75].

5.4 Summary

This chapter described the implementation of MBus and MPQ and evaluated their efficacy in real-world Smart Dust systems. In many ways, the simple existence of operational millimeter-scale systems composed of modular pieces marks the success of the proposed design. The real evaluation, however, will be time. Only time can reveal whether the ideas presented here successfully accelerate the design and implementation of a range of Smart Dust systems. With this thought in mind, the next and final chapter steps back and attempts to consider what else may be required to achieve an ecosystem of Smart Dust—now that we can build these devices, what is required to help deploy them in practice?

[2] Available here: https://github.com/mbus/libmbus

[3] All of the stub functions (e.g. `read_SCL()`) were converted into direct memory accesses assuming a single memory operation MMIO interface.

Chapter 6

Debugging & Bringup

The central focus of this dissertation is the design and implementation of modular Smart Dust systems. The goal of such modularity is to increase the number and diversity of such systems, towards the creation of a Smart Dust ecosystem. A computational ecosystem requires supporting tools. Smart Dust will need mechanisms that aid the development and bringup of new machines. It will also require new introspection paradigms—how does one attach a debugger to systems that fit *inside* of many standard connectors?

This chapter seeks to serve as a starting off point for several of these challenges. It looks at how we might use advanced simulation, coupled with a modern take on in-circuit emulation, to both accelerate the bringup of systems with new and emerging hardware as well as to support debugging in existing hardware. It then looks at some of the challenges of encapsulated systems. This is primarily concerned with the question of how to introspect and debug systems outside of the laboratory environment. We will see how the design decisions outlined in Chapter 4 enable both wireless programming and potentially wireless debugging.

6.1 M-ulator & ICE

One challenge in the development of a new physical computing platform is the variation in the development of each piece of the computing platform. Smart Dust systems are composed of new and novel ultra-low power sensors, memories, processors, and radios. These pieces naturally develop and stabilize at different rates. The benefit of modular design that allows each of these pieces to be worked on independently is also the risk: the realization of a integrated system requires all of pieces. If a system must wait until each modular component is available to perform integration testing, this can dramatically slow the iteration and refinement process for the system as a whole.

One classical means to address this limitation is simulation. Simulation capabilities span a wide range, from circuit-conscious tools, to RTL execution environments, to architecture-conscious cycle-accurate modeling, to possibly looser transaction level modeling [76]. Most simulation, however, focuses on the performance of the CPU core. Recent efforts have started

to look at adapting classical architectural simulation techniques to become platform-aware, multi-component simulations, but these are still in early stages and require significant effort to implement simulations of peripherals [77]. Instead, the traditional approach in platform design is In-Circuit Emulation (ICE) [78].

Decades ago, the usual approach to ICE was to physically remove the CPU core and then to attach leads to a hardware simulator in its place. With the advent of more advanced in-core debugging faculties, however, ICE has shifted. Today, external JTAG controllers can trace, pause, rewind, and overwrite CPU execution. However, these advanced debugging faculties incur non-trivial overhead. Indeed, for the ARM Cortex-M0 implemented in Chapter 5, debugging support would have required nearly as much die area as the core itself! Furthermore, JTAG-based debugging requires physical connection to the core, which is impractical to support for many Smart Dust configurations. For these reasons, traditional JTAG-based debugging is not included in the Smart Dust implementations of this dissertation. This motivates re-considering historical ICE techniques for early-stage bringup when invasive hardware (i.e. replacing the core with a simulator) is viable, and later the development of new in-situ debugging techniques for when hardware-based ICE is not an option.

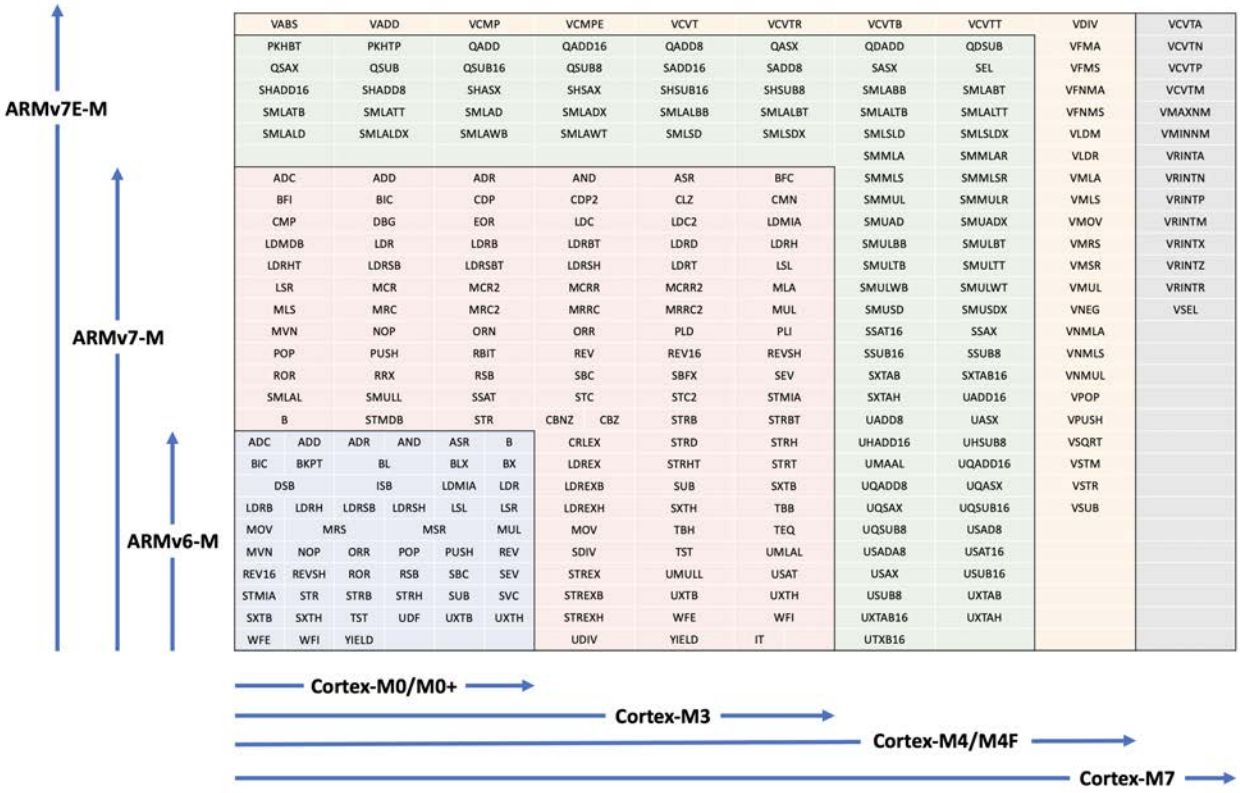
The Case for a Modular CPU Simulator

Instruction set architectures (ISAs) are often thought of as fairly static things. In reality, however, they see continuous, progressive evolution. In performance cores, each major processor iteration from Intel adds new extensions to the architecture. In the embedded space, this process trends both directions. Cores do add extensions that improve performance, but they also sometimes reduce capability to reduce impacts such as energy overhead or die-area demand. Indeed, after the success of the Cortex-M3 and later Cortex-M1, ARM released the further reduced Cortex-M0, their minimalist core, which was the quickest licensed processor product in ARM history [79]. Figure 6.1 captures some of the heterogeneity in just the embedded profile of modern ARM cores. As a consequence of this diversity, it is insufficient to simply build an "ARM simulator." Rather, a simulator should be able to easily adapt between the different ISA flavors and capabilities, to better match the hardware ecosystem.

Design of a Modular CPU Simulator

One view of an instruction set architecture is as a 16- or 32-bit instruction address space, where encoded instructions map onto operations. Instead of a central core that implements each ISA variant, with this view it is instead possible to create a registration-based, modular core. Software modules that make up parts of an ISA expose hooks that express what types of instructions they are capable of decoding and executing:

```
__attribute__ ((constructor))
static void register_opcodes_arm_v6m_mov(void) {
  register_opcode_mask_16(0x4600, 0xb900, mov_reg_t1);
```



Graphic adapted from "ARM Cortex-M for Beginners" [80].

Figure 6.1: **The ARM microcontroller-class ISA hierarchy** The ARM ISA exposes a wide tradeoff space across complexity and capability for CPU designers. The simplest cores facilitate basic data processing and peripheral control tasks. Progressively more advanced cores add support for more advanced data processing, complex bit field manipulations, SIMD, and floating point operations.

During startup, these registrations run before the simulator begins operation. Composing a core then requires simply linking in each of the instructions that are valid for a given ISA.

In the case of ARM in particular, this concept can actually extend further. The example above is for the `t1` encoding of the `mov_reg` (Move Register) operation. This is the only supported encoding for Cortex-M0's, but the Cortex-M3 profile adds `t2`, `t3`, and `t4` encodings (for things such as access to higher-numbered registers or control of status flags). From a functional modeling perspective, there is one Move Register operation that captures the capability of all of these encodings. The compute core then can be made of a generic Move Register operation, optimized for simulation performance and shared among all implementations, while the design-time selection of ISA becomes minimal decoding shims:

```
static void mov_reg_t1(uint16_t inst) {
  uint8_t rd =   (inst &  0x7);
  uint8_t rm =   (inst & 0x78) >> 3;
  uint8_t  D = !!(inst & 0x80);

  rd |= (D << 3);
  uint8_t setflags = false;

  return mov_reg(rd, rm, setflags);
}

static void mov_reg_t3(uint32_t inst) {
  uint8_t       rm =  inst & 0xf;
  uint8_t       rd = (inst >>  8) & 0xf;
  uint8_t setflags = (inst >> 16) & 0x1;

  return mov_reg(rd, rm, setflags);
}
```

Adding Peripherals to a Modular CPU Simulator

Much of what makes embedded processing chips interesting is not the processing core itself, but peripherals in its memory map. In hardware, these peripherals are often designed and implemented as isolated modules, connected by the shared memory bus. Simulation can again mimic this design point and capitalize on a registration-based philosophy to build a modular core:

```
__attribute__ ((constructor))
void register_memmap_ram(void) {
  union memmap_fn mem_fn;

  mem_fn.R_fn32 = ram_read;
  register_memmap("RAM", false, 4, mem_fn, RAMBOT, RAMTOP);
  mem_fn.W_fn32 = ram_write;
  register_memmap("RAM", true, 4, mem_fn, RAMBOT, RAMTOP);
}
```

In this example, even something as fundamental to a core as its memory is attached via registration to the memory bus.

Future Work: Design of a Modular Platform Simulator

The simulator presented in this document is capable of simulation of a complete CPU core and its on-die, integrated peripherals. The simulator also exposes I/O, both for interfacing with real hardware as part of the in-circuit emulation discussed next as well as other software simulators. The question of how to integrate multiple simulators into a cohesive and complete platform simulator remains yet unaddressed by this work. However, once again, a registration-based architecture seems like an appropriate design point. Much as on-die peripherals share a common internal memory bus, the chips that make up a Smart Dust platform now share the common interconnect bus introduced in this thesis. This can again be captured by an interconnect simulator and dispatch core. For simulation, however, this interconnect point cannot only be a message bus, but also must expose primitives that express a joint understanding of time, since the modules under simulation no longer share a common clock source. ICE necessarily forgoes this, as the simulated core cannot keep pace with real hardware anyway, but for system modeling such time management would be of great value.

Narrow Interfaces Enable Record & Replay

The majority of the runtime implementation of any simulated core under this architecture exists in the dynamically loaded modules. Still, these modules do need a common API back to the actual core. In practice, this reduces to the following primitives:

`tick`	Entry point called to compute results from this cycle
`register_{read,write}`	Access a "register-like" object
`memory_{read,write}`	Access a "memory-like" address
`state_{read,write}`	Access custom, local, architectural state

One interesting challenge in modeling hardware is properly capturing parallel execution. The simulator core follows a tick/tock design, where all results for a cycle are computed during the tick, but not written until the tock. What is critical in this interface, is that only the tick is exposed to individual modules. The tock is handled by the core, which ensures that no writes complete until all modules have computed using the correct values for the active cycle. This allows the simulator core to execute module ticks in any arbitrary order—indeed, the simulator will actually spawn local threads if appropriate and execute simulated modules in parallel, unbeknownst to the modules themselves. For this tick/tock operation to work correctly, all access to any potentially shareable architectural state must go through the remaining listed interfaces.

This narrow interface forms the foundation of an (optional—runtime performance tradeoff) deterministic record and replay system for the simulator. Because all architectural state goes through these narrow interfaces, the core can easily track all hardware updates from every cycle. Rewinding then simply requires storing the previous value as well in this data structure. Unlike many CPU-centric simulator designs, I/O between integrated peripherals is implicitly captured in this design point, since the peripherals run under the same execution context and are restricted to the same narrow, tracking-capable interface.

M-ulator as a Pedagogical Tool

Another advantage of modular design is the parallels to educational philosophy. When introducing complex topics, such as the operation of an embedded core or integrated System-on-Chip (SoC) platform, instruction often begins with simplified views that focus on one complex piece. A modular simulator enables empirical lab work that mirrors this same philosophy. When learning encoding, students can modify just the registration of one instruction and see how it changes the rest of the core's operation, with no need to understand any of the rest of the system.

As a case study in the value of such a design, in the fall of 2012, we undertook the audacious proposal of having an entire class of undergraduate students work in a single, shared repository to implement a real-world ISA largely from scratch—in one month. The initial machine supported only two simple instructions: move immediate and unconditional branch. This is just enough to allow the execution of code that demonstrates easily observable architectural changes and runs forever. Each student was independently assigned one operation to implement (e.g. add register), one encoding for a different operation (e.g. load word from lower registers with immediate offset), and one instruction they were responsible for verifying the correctness of. Alongside this was opaque, pre-compiled, instructor-validated implementations that served as a reference. In this way, students could compose a simulator made of a mixture new parts from the class as they were developed with parts that were known to work such that they could test their own implementations. Figure 6.2 shows how this was presented to students, who were tasked to write both operations and testbenches. In the end, the capability for independent progress coupled with shared accountability resulted a core that successfully executed every instructor supplied testbench (and, the student testbenches even found some bugs in the instructor solution!). As the course continued, the simulator was later used to demonstrate the principles of memory-mapped peripherals, as well as the foundation of an ethical-hacking project as a group component of the final exam.

M-ulator as a Debugging Tool

Hardware becomes available in stages. First, there is a high-level expression of what a piece of hardware will do—a functional model. Then, there is RTL and possibly synthesis, a yet-to-be-fabricated, but precise representation of how hardware will operate. Finally, there is the actual, realized hardware itself. As a functional simulator, M-ulator provides an independent reference implementation of what hardware is expected to do.

Cutting edge, experimental hardware holds the risk of truly unexpected bugs. As an example, as part of a refactor to begin optimizing the code that would run on the Smart Dust systems for size, several variables holding only Boolean true/false values were converted from `uint32_t` to `uint8_t`. With only this change, some of the variables could no longer be set to true. In some cases, trying to set one variable to true would actually set a different variable true. Both the realized silicon and RTL simulation exhibited the same behavior. Could this be a compiler bug, what could cause such behavior? Running the same code on the M-ulator,

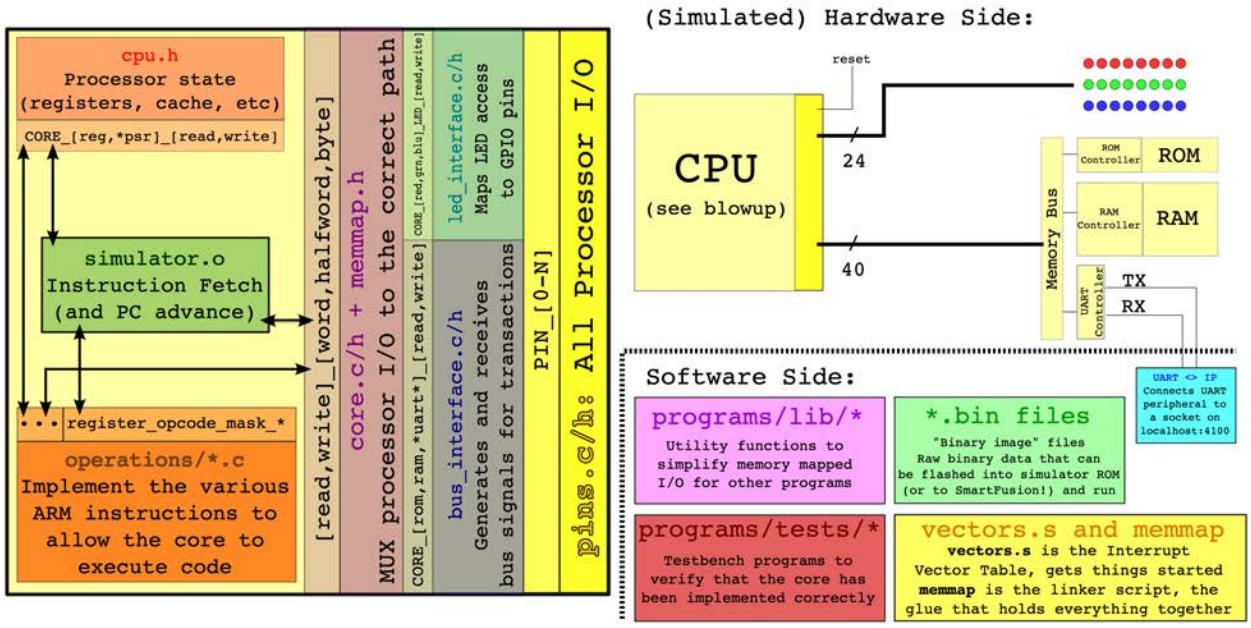


Figure 6.2: **High-level design of M-ulator.** The architecture of M-ulator mirrors that of real hardware. This allows for runtime modularity, where individual components can be implemented either via software emulation or as pass-throughs to real hardware. It further serves a pedagogical goal, as it allows laboratory and project work to explore and implement normally tightly-integrated components in isolation.

however, each variable was set and read back correctly. Different execution traces between M-ulator and hardware allowed rapid confirmation of a hardware bug.

As the design of M-ulator allows for introspection of architectural state, in this case, it was also able to assist in identifying the bug. The two implementations performed identically until one cycle where the RTL implementation wrote to a different memory address than the M-ulated version. The instruction and register contents were the same in both machines, yet the memory controller in the RTL version had a slightly different address. This pointed to some error between the CPU core and the memory controller—a narrow search space. Because both the MBus controller and the CPU can instantiate transactions on the memory bus, they must go through an arbiter to access the memory bus. The MBus implementation was only capable of full, word-aligned access and therefore simply tied the bottom two address bits to zero. The CPU, however, was now attempting to issue single byte memory bus transactions where the bottom two bits suddenly hold significance. With the confidence that there was an error in the hardware, and a narrow search space, this entire debugging process took less than an afternoon.

In-Circuit Emulation & the ICE Board

Due to the severe resource constraints of Smart Dust systems, the physical CPU chip has no on-board debugging faculties. When tracking down system integration bugs, however, a debugger is an invaluable tool. The simulator is effectively capable as operating as a CPU chip in a Smart Dust system—it can run the same software and should exhibit the same behavior. Using this capability, however, requires the ability to interface the simulated core with the rest of the chips in a Smart Dust system.

Direct Hardware Mapping is Too Slow

As an initial thought, the simplest hardware interface would simply export each pin described by a module in the simulator to a remote GPIO pin on a partner hardware platform. Such a bit-banged approach would require the simulator to achieve precise timing control over these remote pins, however. When interfacing with hardware, real-world timing constraints must be met. Pure simulation allows for time dilation, the simulator core can slow all modules to run at the same speed relative to one another. The simulator does not control the clocks on remote hardware, however, which means that it must support true real-time operation. This was not a design goal of the simulator, however, which sought functional correctness but not necessarily strict timing.

M-ulation

Bit-banging pins is not the way that a real CPU interfaces with hardware either. Rather, application-specific peripherals handle timing sensitive operations, such as the low-level details of a bus transaction, and the CPU interacts with these peripherals over memory mapped I/O. This insight describes the in-circuit emulation strategy. Figure 6.3 shows the support hardware board, which includes an FPGA capable of implementing arbitrary peripherals. To the simulator, the FPGA is simply another peripheral that hangs off the memory bus of the simulated core. The simulator sees no difference between interfacing with real-world hardware and or another simulated peripheral.

There is one key difference between a hardware peripheral and a simulated peripheral. Simulated peripherals are only permitted to manipulate hardware state following the simulated clock. For this reason, the FPGA does not literally hang off of the simulated memory bus. Instead, there is an interface layer which acts as a peripheral to the simulator and uses a custom protocol specific to and optimized for each peripheral to interact with the FPGA.

Towards Reproducible Execution Traces While the simulator cannot stop time in the real world, it can replay it. This section describes one aspect of the FPGA protocol which is an initial step towards deterministic record and replay of hardware execution events. The core of the idea is an event log that runs between the FPGA and the support peripheral that interfaces with the simulator. If the simulator wishes to rewind and replay events, the support layer could in principle use the timing provided by a real-world event trace from

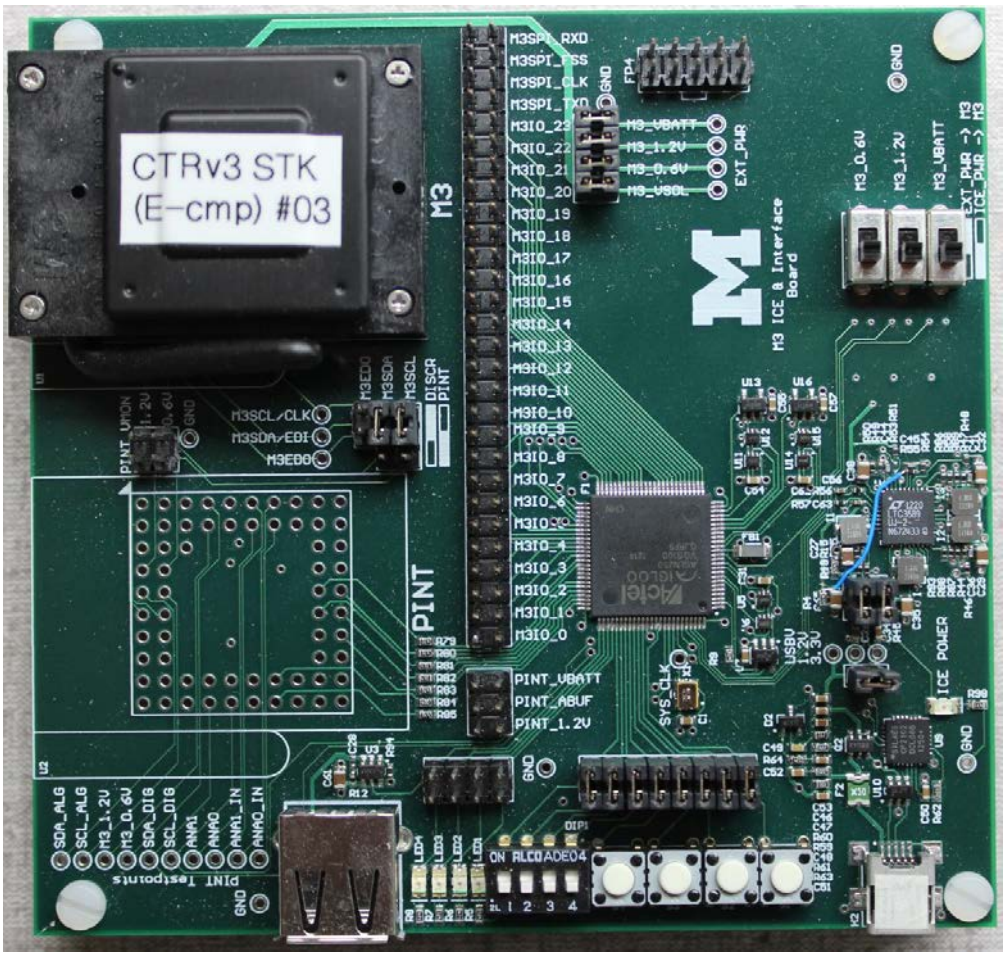


Figure 6.3: **The ICE board.** A custom hardware board that allows flexible hardware access. It supports both traditional in-circuit emulation tasks as well as more general purpose debugging and programming of arbitrary hardware in the greater M3 Smart Dust ecosystem.

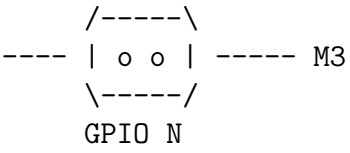
hardware coupled with a model for that hardware peripheral to allow replay of hardware event sequences. As a first step, the hardware and interface work to capture and label a coherent view of events.

Event IDs are an unsigned single byte number. They define a total ordering the events actually occurred in the system. In particular, if a command message is being sent to set GPIO 0 (an output) high at the same time that GPIO 1 (an input) goes high, the ordering will be `Event N: GPIO 1 --> High` then `Event N+1: GPIO 0 --> High`. For message format consistency, event ids are included in both directions of communication, however the field may be safely ignored by the FPGA. The FPGA itself must by definition have some order of I/O events it processed, which are encapsulated by these event ids. The event id of a control message is assigned by the FPGA whenever it actually processes the event and is indicated to the controller via the ACK message.

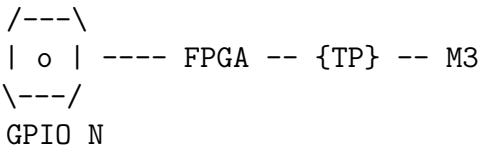
Some decisions in microcontrollers necessarily race. As contrived example, if two GPIO pins are defined as interrupts but are electrically connected in the external circuit, and then line is pulled high, which interrupt fires first? While the decision is arbitrary, there is motivation to define an ordering of events in the system. The events observed by ICE can be replayed in the M-ultaor for debugging, but the is only possible if they can be accurately re-created. In practice, this diverges to two ideas:

Concurrent Events: Two events could be labeled with the same event id, indicating that they occurred too close together in time for the ICE to distinguish them. This would require divergence in the simulator for debugging runs. While conceptually feasible, this is non-trivial and could quickly balloon into an intractable number of concurrent execution paths.

I/O Pass-Through: Currently, the FPGA and the M3 share GPIO pins (Nx2 header):



Instead, this can be rearranged to explicitly pass all I/O through the FPGA:



This has the disadvantage of doubling the number of FPGA I/O pins consumed for each of the M3 I/O pins. It does enable the FPGA to strictly define an order of events. On the existing ICE hardware prototype, there is sufficient hardware available to adapt to this model for one bank of eight I/O pins (e.g. mapping GPIOs 16-23 as FPGA-input only, jumpering over GPIOs 8-15 and having the FPGA drive those, leaving GPIOs 0-7 as the first one).

The ICE Board as General Platform Support Tool

There is additional motivation for not directly hanging the ICE board off of the simulated memory bus. Many use cases for the ICE board do not require the simulator at all. As a general purpose hardware platform, the ICE board is useful as a support and interface tool for the greater M3 Smart Dust ecosystem. For this reason, the interface to the ICE board is written as a general purpose Python library.[1]

As MBus frontends and support boards are not yet prevalent devices, the board can operate as a general purpose MBus interface. Lightweight tools allow for snooping of MBus

[1]Available here: https://github.com/mbus/m3-python. Also included here is a command-line scripting environment for direct interfacing with the ICE board as well as a simulator for the ICE board itself, to allow M-ulation without ICE hardware.

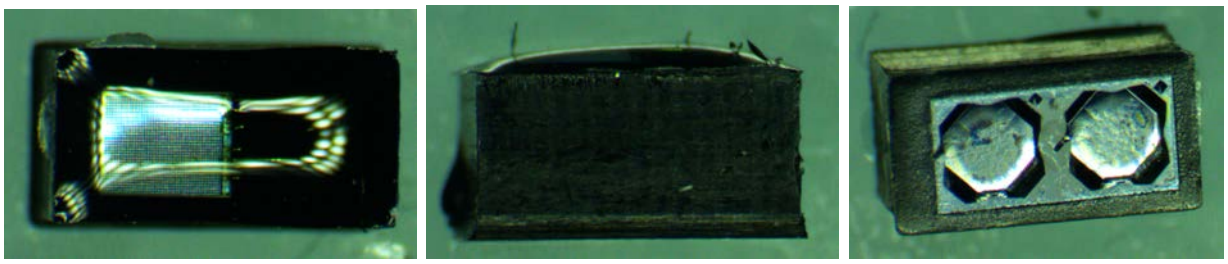


Figure 6.4: **A pressure sensor encapsulated in bio-compatible epoxy.** Deployment considerations for Smart Dust systems limit the capacity to physically interface with devices.

transactions or programmable interaction with MBus events. The FPGA image and supporting libraries also can support an I^2C peripheral, GPIO peripherals, and several other specialty hardware peripherals. For systems that are physically accessible, such as the CPU chip seen in debug packaging in Figure 6.3, MBus transactions sent via the ICE board are actually the primary means of programming. This is yet one more advantage of the system bus design, wherein programming is simply an MBus DMA operation that writes instruction memory.

6.2 Encapsulated Interactions

What happens once we leave the realm of debug systems and debugging-optimized packaging? Consider for example the system shown in Figure 6.4. This is a system that has been wholly encapsulated in a bio-compatible epoxy for implantation. This is not a system that can be plugged into to program or debug.

Wireless Interactions

The natural solution to this problem is to communicate wirelessly with the Smart Dust device. Here, the resource constraints rear up again in force. Indeed, the story has actually come full circle to the original Smart Dust efforts from the 1990s, where much of the research effort was dedicated to addressing the communication problem [1].

RF Communication and the Idle Listening Problem

There has been significant advancement in the capability of low-power RF-based communication. Today, it is feasible to support short bursts of two-way RF-based communication on a cubic-millimeter energy budget. The challenge, however, lies in the time between communications. The viability of RF-based communication, like much of the Smart Dust system, hinges on the ability to duty cycle the radio frontend. This line of thinking leads, of course, to the well-known idle-listening problem [30]—when should the radio turn on? At Smart Dust energy budgets, periodic wakeup of the most energy-intensive element simply to learn there is no communication occurring is not a wise design point.

A Return to Visible Light Communication

Again, therefore, we look back to the origins of Smart Dust, and find this problem too was both anticipated and solved [24]. Visible Light Communication receives requires significantly less energy. The systems demonstrated earlier all include a 695 pW always-on optical frontend [81]. This hardware component is capable of both acting as a wakeup trigger as well as receiving arbitrary, encoded data.

Remote Programming and Debugging with MPQ

The salient question for this dissertation is how to meaningfully integrate this visible light communication frontend with the rest of the encapsulated system? The answer continues to demonstrate the generality and flexibility of the generic and general purpose central bus interface: it is simply one more device on the system bus. The optical link directly writes arbitrary messages onto the system bus, with no requirements for any local decision-making or computational capability in the optical receiver (beyond a few bits of buffering).

To begin, this interface allows for remote programming of the system. Recall, that programming systems with MPQ can be achieved by simply issuing a DMA transfer to the program memory of a CPU chip. By issuing a streaming transaction, a remote endpoint can upload complete program images directly to an encapsulated system. This design eschews traditional complexities such as the transient need for space for both the existing bootloader that is receiving the program and the new program that is being loaded. While a transaction may be interrupted or corrupted, unlike the bootloader update scenario, this does not permanently corrupt the machine. A new transaction can simply try again, issuing a fresh DMA that overwrites the corrupt image.

Supporting arbitrary messages allows for remote debugging as well. Here, a debugging system capitalizes on the ability of the system bus to exert operations on any node in the system. A remote endpoint can issue transactions that activate sensors, trigger internal test procedures, and even write results to radio memory and trigger packets to enable bidirectional communication during the debugging session.

As a thought experiment, one could even consider the implementation of an approximation of remote JTAG. Assuming a small amount of program memory is unused and available, a miniature debugger instance could be loaded (via streaming DMA). Breakpoints could be simulated by overwriting key instructions with jumps to the loaded debugging instance.

Even single-step operation is possible. Here, one would replace an instruction with a break into the micro-debugger routine. The debugger would then replace the jump into itself with the original instruction and overwrite the next instruction in the stream with a call back to itself. Certain overwritten instructions, such as branches, would require the debugger to actually simulate the execution of the instruction to properly complete this operation. Loading a simulator into scratch space on the device is intractable, but instead the single-stepping mode could expand to full M-ulation. Upon first entry, the trap routine reports all hardware state to a remote simulator instance. Subsequently, the loop becomes:

offload the newly overwritten instruction to the simulator and then receive back a series of hardware state updates to issue via micro-ops streamed into a small execution buffer in the trap; these micro-ops would include restoring the executed instruction in main memory and loading the next instruction to overwrite. In this way, a remote debugger and simulator instance with no a priori knowledge of the code on the Smart Dust node could implement full single-step debugging. Such a design is certainly not efficient, but it does demonstrate the power and expressivity of the system bus interface: it enables full-featured debugging capabilities with zero hardware modification.

6.3 Summary

This chapter looks at some of the initial questions that come up as millimeter-scale computing moves beyond the creation and composition of Smart Dust devices. These are the pragmatic questions that ask how such devices will actually be deployed and used—how to load code onto Dust, how to debug Dust? Here again, the modular, composable design shines, only now that modularity is extended outside the physical envelope of the system. Conceptually, these interfaces allow the dynamic interposition of arbitrary modules into encapsulated systems. This is a powerful, but optimistic, primitive. This dissertation answers how to enable this class of computing. It does not answer how to secure Smart Dust or how to secure the world in the face of its development.

Chapter 7

Conclusion

Embedded systems are interesting because they are embedded in the physical world. The capability to put computation and intelligence into the environment expands the reach of the digital world into the physical world. Physically shrinking computing expands this reach further by increasing the number and diversity of scenarios into which computing can reasonably be embedded. By its nature, however, each embedded context is different. This demands a significantly greater diversity in the design and capability of individual embedded systems as compared to traditional computing domains.

A ring topology is, perhaps surprisingly, not just appropriate for resource-constrained systems but key to enabling composable design. A clockless, "shoot-through" ring can alleviate much of the overhead and complexity associated with traditional ring-based architectures. For specialized, embedded devices, the tight integration required by a ring does not generally introduce meaningful fragility—what use is a wireless sensor node without a working sensor chip? When this computing class is able to evolve back towards generality, the use of an isolated, logic-free "shoot-through" frontend mitigates risk from failures and enhances robustness, as it allows the ring-based system to operate despite other failures.

Composable design is achievable for and useful to the millimeter-scale computing class. There are overheads to modularity and new challenges in the composition of pitch black silicon, but the careful design of an interconnect and its interfaces enables viable modular Smart Dust. While this result makes a library-of-parts model possible, it does not will such a library into existence, nor then is the resultant ecosystem of millimeter-scale systems yet realized. There remain numerous key challenges to the widespread emergence of the Smart Dust computing class, such as physical packaging, security models, localization, and decommissioning. It is my sincerest hope, however, that the work presented in this dissertation can accelerate all of these efforts, and help to enable the advent of a new class of computing.

Bibliography

[1] Kris Pister. *SMART DUST: Autonomous sensing and communication in a cubic millimeter*. Accessed 2020. 2001. URL: https://people.eecs.berkeley.edu/~pister/SmartDust/.

[2] Gregory Chen et al. "A Cubic-millimeter Energy-autonomous Wireless Intraocular Pressure Monitor". In: *2011 IEEE International Solid-State Circuits Conference*. 2011, pp. 310–312.

[3] Gregory K. Chen. "Power Management and SRAM for Energy-Autonomous and Low-Power Systems". http://hdl.handle.net/2027.42/86387. PhD thesis. University of Michigan, 2011.

[4] Yoonmyung Lee. "Ultra-Low Power Circuit Design for Cubic-Millimeter Wireless Sensor Platform". http://hdl.handle.net/2027.42/91438. PhD thesis. University of Michigan, 2012.

[5] Gyouho Kim. "Ultra-Low Power Optical Interface Circuits for Nearly Invisible Wireless Sensor Nodes". http://hdl.handle.net/2027.42/110399. PhD thesis. University of Michigan, 2014.

[6] Taekwang Jang. "Circuit and System Designs for Millimeter Scale IoT and Wireless Neural Recording". http://hdl.handle.net/2027.42/140970. PhD thesis. University of Michigan, 2017.

[7] Wootaek Lim. "Ultra-Low Power Circuit Design for Miniaturized IoT Platform". http://hdl.handle.net/2027.42/145862. PhD thesis. University of Michigan, 2018.

[8] Li-Xuan Chuo et al. "A 915 MHz Asymmetric Radio Using Q-enhanced Amplifier for a Fully Integrated $3 \times 3 \times 3\,\mathrm{mm}^3$ Wireless Sensor Node with 20 m Non-Line-of-Sight Communication". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 132–133.

[9] Yao Shi et al. "A $10\,\mathrm{mm}^3$ Inductive Coupling Radio for Syringe-Implantable Smart Sensor Nodes". In: *IEEE Journal of Solid-State Circuits* 51.11 (2016), pp. 2570–2583.

[10] Mingoo Seok et al. "The Phoenix Processor: A 30 pW platform for sensor applications". In: *2008 IEEE Symposium on VLSI Circuits*. 2008, pp. 188–189.

[11] Qing Dong et al. "A 1 Mb Embedded NOR Flash Memory with 39 µW Program Power for mm-scale High-temperature Sensor Nodes". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 198–199.

[12] Gyouho Kim et al. "A 467nW CMOS Visual Motion Sensor with Temporal Averaging and Pixel Aggregation". In: *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 2013, pp. 480–481.

[13] Minchang Cho et al. "A $6 \times 5 \times 4\,\mathrm{mm}^3$ General Purpose Audio Sensor Node with a 4.7 µW Audio Processing IC". In: *2017 Symposium on VLSI Circuits*. 2017, pp. C312–C313.

[14] Dongsuk Jeon et al. "An Implantable 64 nW ECG-monitoring Mixed-signal SoC for Arrhythmia Diagnosis". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 416–417.

[15] Sechang Oh et al. "A 2.5 nJ Duty-cycled Bridge-to-digital Converter Integrated in a $13\,\mathrm{mm}^3$ Pressure-sensing System". In: *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. 2018, pp. 328–330.

[16] Kaiyuan Yang et al. "A 0.6 nJ -0.22/+0.19°C Inaccuracy Temperature Sensor Using Exponential Subthreshold Oscillation Dependence". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 160–161.

[17] Yoonmyung Lee et al. "A Modular $1\,\mathrm{mm}^3$ Die-Stacked Sensing Platform with Low Power $\mathrm{I}^2\mathrm{C}$ Inter-die Communication and Multi-Modal Energy Harvesting". In: *IEEE Journal of Solid-State Circuits* 48.1 (Jan. 2013), pp. 229–243.

[18] Gordon Bell. "Bell's Law for the Birth and Death of Computer Classes". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 86–94. ISSN: 0001-0782. DOI: 10.1145/1327452.1327453. URL: http://doi.acm.org/10.1145/1327452.1327453.

[19] Gordon Bell. "Bell's Law for the Birth and Death of Computer Classes: A Theory of the Computer's Evolution". In: *IEEE Solid-State Circuits Society Newsletter* 13.4 (2008), pp. 8–19.

[20] Victoria Rozycki. *The Computer Musuem*. http://tcm.computerhistory.org/. (Accessed 2020). 2015.

[21] Michael D. Scott, Bernhard E. Boser, and Kristofer S.J. Pister. "An Ultralow-energy ADC for Smart Dust". In: *Solid-State Circuits, IEEE Journal of* 38.7 (2003), pp. 1123–1129. ISSN: 0018-9200. DOI: 10.1109/JSSC.2003.813296.

[22] Richard Yeh Robert, Robert A. Conant, and Kristofer S. J. Pister. "Mechanical Digital-To-Analog Converters". In: *Proceedings of the Tenth International Conference on Sensors and Actuators (Transducers '99)*. 1999, pp. 7–10.

[23] Matthew Last and Kristofer S.J. Pister. "2-DOF Actuated Micromirror Designed for Large DC Deflection". In: MOEMS'99. 1999.

[24] Patrick B. Chu et al. "Optical communication using micro corner cube reflectors". In: *Micro Electro Mechanical Systems, 1997. MEMS '97, Proceedings, IEEE., Tenth Annual International Workshop on*. 1997, pp. 350–355. DOI: 10.1109/MEMSYS.1997.581852.

[25] Brett A. Warneke and Kristofer S.J. Pister. "An Ultra-low Energy Microcontroller for Smart Dust Wireless Sensor Networks". In: *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*. 2004, 316–317 Vol.1. DOI: 10.1109/ISSCC.2004.1332721.

[26] Seth Edward-Austin Hollar. "COTS Dust". https://people.eecs.berkeley.edu/~pister/publications/dissertations/hollarms2000.pdf. MA thesis. University of California, Berkeley, Dec. 2000.

[27] Joe M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. "Next Century Challenges: Mobile Networking for "Smart Dust"". In: *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. MobiCom '99. Seattle, Washington, USA: ACM, 1999, pp. 271–278. ISBN: 1-58113-142-9. DOI: 10.1145/313451.313558. URL: http://doi.acm.org/10.1145/313451.313558.

[28] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. "Emerging challenges: Mobile networking for "Smart Dust"". In: *Journal of Communications and Networks* 2.3 (2000), pp. 188–196.

[29] Miklós Maróti et al. "The Flooding Time Synchronization Protocol". In: *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. SenSys '04. Baltimore, MD, USA: Association for Computing Machinery, 2004, pp. 39–49. ISBN: 1581138792. DOI: 10.1145/1031495.1031501. URL: https://doi.org/10.1145/1031495.1031501.

[30] Joseph Polastre, Jason Hill, and David Culler. "Versatile Low Power Media Access for Wireless Sensor Networks". In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. 2004, pp. 95–107.

[31] Pei Zhang et al. "Hardware Design Experiences in ZebraNet". In: *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. SenSys '04. Baltimore, MD, USA: Association for Computing Machinery, 2004, pp. 227–238. ISBN: 1581138792. DOI: 10.1145/1031495.1031522. URL: https://doi.org/10.1145/1031495.1031522.

[32] Jonathan W. Hui and David E. Culler. "IP is Dead, Long Live IP for Wireless Sensor Networks". In: *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. SenSys '08. Raleigh, NC, USA: Association for Computing Machinery, 2008, pp. 15–28. ISBN: 9781595939906. DOI: 10.1145/1460412.1460415. URL: https://doi.org/10.1145/1460412.1460415.

[33] Filip Maksimovic et al. "A Crystal-Free Single-Chip Micro Mote with Integrated 802.15.4 Compatible Transceiver, sub-mW BLE Compatible Beacon Transmitter, and Cortex M0". In: *2019 Symposium on VLSI Circuits*. 2019, pp. C88–C89. DOI: 10.23919/VLSIC.2019.8777971.

[34] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Volume 1. The Power of Modularity*. 2000.

[35] Mali Mahalingam. "Thermal Management in Semiconductor Device Packaging". In: *Proceedings of the IEEE* 73.9 (1985), pp. 1396–1404. DOI: 10.1109/PROC.1985.13300.

[36] iFixit. *iPhone X Teardown – iPhoneX-xray.jpg*. https://www.ifixit.com/Guide/Image/meta/iLidRyQIbJbNnTO2. Accessed 2019. Nov. 2017.

[37] iFixit. *Apple Watch Series 4 Teardown – watch-x-ray.jpg*. https://www.ifixit.com/Guide/Image/meta/aUum3m5axOUDWMXo. Accessed 2019. Sept. 2018.

[38] Bruno Scrosati. "History of Lithium Batteries". In: *Journal of Solid State Electrochemistry* 15.7 (July 2011), pp. 1623–1630. ISSN: 1433-0768. DOI: 10.1007/s10008-011-1386-8. URL: https://doi.org/10.1007/s10008-011-1386-8.

[39] Benton H. Calhoun and David Brooks. "Can Subthreshold and Near-Threshold Circuits Go Mainstream?" In: *IEEE Micro* 30.4 (2010), pp. 80–85. DOI: 10.1109/MM.2010.60.

[40] Bo Zhai et al. "Theoretical and Practical Limits of Dynamic Voltage Scaling". In: *Proceedings. 41st Design Automation Conference*. 2004, pp. 868–873.

[41] Arun Raghavan et al. "Computational Sprinting". In: *IEEE International Symposium on High-Performance Comp Architecture*. 2012, pp. 1–12. DOI: 10.1109/HPCA.2012.6169031.

[42] Michael B. Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *IEEE Micro* 33.5 (2013), pp. 8–19. DOI: 10.1109/MM.2013.90.

[43] Ye-Sheng Kuo et al. "MBus: A 17.5 pJ/bit/chip Portable Interconnect Bus for Millimeter-scale Sensor Systems with 8 nW Standby Power". In: *Custom Integrated Circuits Conference (CICC), 2014 IEEE Proceedings of the*. Sept. 2014, pp. 1–4. DOI: 10.1109/CICC.2014.6946046.

[44] Pat Pannuto et al. "MBus: An Ultra-Low Power Interconnect Bus for Next Generation Nanopower Systems". In: *Proceedings of the 42nd International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon, USA: ACM, June 2015.

[45] Pat Pannuto et al. "MBus: A System Integration Bus for the Modular Micro-Scale Computing Class". In: *IEEE Micro: Special Issue on Top Picks from Computer Architecture Conferences* 36.3 (May 2016), pp. 60–70. ISSN: 0272-1732. DOI: 10.1109/MM.2016.41.

[46] Inhee Lee et al. "MBus: A Fully Synthesizable Low-power Portable Interconnect Bus for Millimeter-scale Sensor Systems". In: *Journal of Semiconductor Technology and Science* 16.6 (Dec. 2016), pp. 745–753. DOI: 10.5573/JSTS.2016.16.6.745.

[47] Sechang Oh et al. "IoT2 – The Internet of Tiny Things: Realizing mm-Scale Sensors through 3D Die Stacking". In: *2019 Design, Automation Test in Europe Conference Exhibition*. DATE'19. Mar. 2019, pp. 686–691. DOI: 10.23919/DATE.2019.8715201.

[48] David Blaauw et al. "IoT Design Space Challenges: Circuits and Systems". In: *Proceedings of the 2014 IEEE Symposium on VLSI Technology (VLSI'14)*. Honolulu, Hawaii, USA, June 2014.

[49] Gyouho Kim et al. "A Millimeter-Scale Wireless Imaging System with Continuous Motion Detection and Energy Harvesting". In: *VLSI Circuits (VLSIC), 2014 Symposium on*. Honolulu, Hawaii, USA, June 2014.

[50] Pat Pannuto et al. "Demo: Ultra-constrained Sensor Platform Interfacing". In: *Proceedings of the 11th International Conference on Information Processing in Sensor Networks*. IPSN '12. Beijing, China: ACM, 2012, pp. 147–148. ISBN: 978-1-4503-1227-1. DOI: 10.1145/2185677.2185721. URL: http://doi.acm.org/10.1145/2185677.2185721.

[51] NXP. *I^2C-bus specification Rev. 6*. http://www.nxp.com/documents/user_manual/UM10204.pdf. Apr. 2014.

[52] *System Management Bus BIOS Interface Specification*. http://smbus.org/specs/smbb10.pdf.

[53] Uwe Kiencke, Siegfried Dais, and Martin Litschel. "Automotive Serial Controller Area Network". In: *SAE Technical Paper 860391* (1986). DOI: 10.4271/860391.

[54] Saed G. Younis. "Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic". http://hdl.handle.net/1721.1/7058. PhD thesis. Massachusetts Institute of Technology, June 1994.

[55] V.S. Sathe et al. "Resonant-Clock Design for a Power-Efficient, High-Volume x86-64 Microprocessor". In: *Solid-State Circuits, IEEE Journal of* 48.1 (Jan. 2013), pp. 140–149. ISSN: 0018-9200. DOI: 10.1109/JSSC.2012.2218068.

[56] MIPI Alliance. *MIPI Overview*. https://www.mipi.org/about-us. Accessed 2020. 2020.

[57] MIPI Alliance. *Specification for I3C Basic*. https://www.mipi.org/specifications/i3c-sensor-specification. Accessed 2020. July 2018.

[58] Cristian Ciocan. "The Domestic Digital Bus system (D2B) – A Maximum of Control Convenience in Audio Video". In: *Consumer Electronics, IEEE Transactions on* 36.3 (Aug. 1990), pp. 619–622. ISSN: 0098-3063. DOI: 10.1109/30.103182.

[59] Mercedes-Benz USA. *Domestic Digital Bus (D2B)*. http://www.mercedestechstore.com/pdfs/416_Telematics/416HOD2B(CooksonI)03-09-04.pdf. 2004.

[60] Rajesh Kumar and Glenn Hinton. "A Family of 45 nm IA Processors". In: *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*. 2009, pp. 58–59. DOI: 10.1109/ISSCC.2009.4977306.

[61] John W. Poulton et al. "A 0.54 pJ/b 20 Gb/s Ground-Referenced Single-Ended Short-Reach Serial Link in 28 nm CMOS for Advanced Packaging Applications". In: *Solid-State Circuits, IEEE Journal of* 48.12 (Dec. 2013), pp. 3206–3218. ISSN: 0018-9200. DOI: 10.1109/JSSC.2013.2279053.

[62] Mozhgan Mansuri et al. "A Scalable 0.128-to-1 Tb/s 0.8-to-2.6 pJ/b 64-lane Parallel I/O in 32 nm CMOS". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*. Feb. 2013, pp. 402–403. DOI: 10.1109/ISSCC.2013.6487788.

[63] Jason Hill et al. "System Architecture Directions for Networked Sensors". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA, 2000, pp. 93–104. ISBN: 1581133170. DOI: 10.1145/378993.379006. URL: https://doi.org/10.1145/378993.379006.

[64] Arm Limited. *AMBA Specification (Rev 2.0)*. https://static.docs.arm.com/ihi0011/a/IHI0011A_AMBA_SPEC.pdf. Accessed 2020. 1999.

[65] SiFive. *SiFive TileLink Specification*. https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf. Accessed 2020. 2019.

[66] Jerry Crow. "The MS-DOS Memory Environment". In: *SIGICE Bull.* 21.3 (Jan. 1996), pp. 2–16. ISSN: 1558-1144. DOI: 10.1145/226036.226037. URL: https://doi.org/10.1145/226036.226037.

[67] Joshua Adkins et al. "The Signpost Platform for City-Scale Sensing". In: *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN'18. New York, NY, USA: ACM, Apr. 2018.

[68] Simon Srot. *SPI Master Core Specification*. svn://opencores.org/ocsvn/spi@r29. OpenCores.

[69] Richard Herveille. *I²C-Master Core Specification*. svn://opencores.org/ocsvn/i2c@r76. OpenCores.

[70] Texas Instruments. *MSP430F161x Mixed Signal Microcontroller*. http://www.ti.com/product/msp430f1611.

[71] Byte Paradigm. *SPI Storm Datasheet*. http://www.byteparadigm.com/files/documents/ds_SPIStorm.pdf. Dec. 2011.

[72] National Instruments. *NI PCIe-6535 10 MHz Digital I/O for PCI Express*. http://sine.ni.com/nips/cds/view/p/lang/en/nid/205628.

[73] Microsemi. *IGLOO nano Low Power Flash FPGAs*. http://www.actel.com/documents/IGLOO_nano_DS.pdf.

[74] Texas Intruments and Red Hat. *GCC – Open Source Compiler for MSP430 Microcontrollers*. http://www.ti.com/tool/msp430-gcc-opensource.

[75] *Wikipedia: Example of bit-banging the I²C Master protocol*. http://en.wikipedia.org/wiki/I2c#Example_of_bit-banging_the_I.C2.B2C_Master_protocol. Accessed 2015. Nov. 2013.

[76] Lukai Cai and Daniel Gajski. "Transaction Level Modeling: An Overview". In: *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*. 2003, pp. 19–24. DOI: 10.1109/CODESS.2003.1275250.

[77] Yakun Sophia Shao et al. "Co-Designing Accelerators and SoC Interfaces Using Gem5-Aladdin". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016.

[78] Chung-Fu Kao, Hsin-Ming Chen, and Jer Huang. "Hardware-software approaches to in-circuit emulation for embedded processors". In: *IEEE Design & Test of Computers* 25.5 (2008), pp. 462–477.

[79] Joseph Yiu. "Chapter 4 - Architecture". In: *The Definitive Guide to Arm Cortex-M0 and Cortex-M0+ Processors (Second Edition)*. Ed. by Joseph Yiu. Second Edition. Oxford: Newnes, 2015, pp. 87–108. ISBN: 978-0-12-803277-0. DOI: https://doi.org/10.1016/B978-0-12-803277-0.00004-7. URL: http://www.sciencedirect.com/science/article/pii/B9780128032770000047.

[80] Joseph Yiu. *White Paper: ARM Cortex-M for Beginners*. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/white-paper-cortex-m-for-beginners-an-overview-of-the-arm-cortex-m-processor-family-and-comparison. Accessed 2020. Sept. 2016.

[81] Gyouho Kim et al. "A 695 pW Standby Power Optical Wake-up Receiver for Wireless Sensor Nodes". In: *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*. 2012, pp. 1–4. DOI: 10.1109/CICC.2012.6330603.