Personal Statement: The Goals of the Blitz-64 Project

Harry H. Porter III
Portland State University

HHPorter3@gmail.com

18 February 2020

This document describes the goals for the Blitz-64 project. The challenges confronting computer systems in today's world are discussed and I ruminate on the decisions and design process in creating a system to address these issues. A number of opinions and philosophical musings are also included.

Table of Contents

Design Philosophy

I must create a system or be enslaved by another man's;
I will not reason and compare: My business is to create.
— William Blake

General Principles and Goals

The primary design criterion of the Blitz-64 project is **simplicity**. The intent is to create a complete, fully functional computer system that can be understood quickly and easily by any skilled programmer. Although the system has evolved over time, this goal remains number one. Simplicity helps to achieve the other goals.

The second design criterion is **reliability.** Once completed and deployed, a computer system should continue to work correctly. We primarily target software failures and always make our design choices in favor of eliminating bugs. The goal should always be complete **eradication of bugs and unpredictable behavior**. Repeatability and **resistance to software rot** is crucial to achieving reliability.

A third criterion is to **increase security** and **improve resistance to malware**. The Blitz-64 project is specifically targeted at **applications that require high-reliability** and **malware resistance**.

In moving toward these goals, Blitz-64 places an enormous **emphasis on catching and reporting errors.** The tradeoff is always performance versus execution speed. I believe we should choose slower systems that work correctly over faster systems that do not. Those who are willing to trade correctness for speed should go elsewhere; Blitz-64 is not for you.

Another design criterion for KPL is to create a programming language that facilitates **readability of programs.** As a consequence, the syntax emphasizes readability, at the expense of terseness and ease of typing. Increasing the **clarity of code** clearly helps in our goals of simplicity and reliability. **Software correctness** is critical and a new language designed explicitly for this objective is part of the project.

Of course, tradeoffs between reliability and execution efficiency cannot be avoided. Traditionally, low-level languages like C and C++ made design choices in favor of efficiency, leaving high-reliability to interpreted languages. Within the class of systems programming languages, KPL puts greater emphasis on reliability and fault-tolerance than familiar languages.

The Greater Problem

Okay, that was some nice boilerplate. The usual clichés, bland banalities, and trite bromides.

The fact is, **the sheer complexity of modern software systems terrifies me**.

Nobody can understand a modern computer. The device in your pocket holds millions of lines of code, some of it written in the 1970s[1]. No single person can understand such complicated artifacts. No one can come close to full comprehension, even with a Computer Science Ph.D.

The devices in our pockets have a remarkable amount of functionality, but they have become **black boxes**. When something goes wrong with the software, it **cannot be fixed or repaired**. Often, we can't even understand the problem. **We no longer understand the technology that we have become totally dependent on.**

The Blitz-64 project really has this goal:

I want a computer system that a single individual can fully understand.

More specifically, I want a computer system that I, myself, can understand. My goal is to design and implement such a system *alone*. Furthermore, my goal is to code all software from scratch, **not relying on preexisting software**. Obviously, this is ambitious.

We have learned a lot about how computer systems can be designed, about what works best, and how to do things right. But our **existing digital infrastructure** rests on a foundation of **work done in the distant past**, before we learned many of the lessons that are now taught to beginning programmers. Your phone is running

[1] Android has 12 million lines of code. [`www.visualcapitalist.com/millions-lines-of-code`]

legacy software which was first designed in the 1970s. While this remarkable software was modern and hugely innovative when it first appeared, we now know a lot more.

Much of **the software that our society relies on**—the software lying below all the complex apps we use, as well as the digital infrastructure that our lives depend on—**is simply ancient** and, by modern standards, grody.

The Blitz-64 project includes:

a completely **new computer architecture**
a completely **new programming language**
a completely **new set of software tools**
a completely **new operating system kernel**
full and complete **documentation**

The Blitz-64 project includes a completely **new computer architecture**. Blitz-64 includes a completely **new programming language**. Blitz-64 includes a completely **new set of software tools**. Blitz-64 includes a completely **new operating system kernel**, on which these tools run. Blitz-64 includes a **full and complete description in English** documents.

The ambition of the Blitz-64 project is to **create an entire computer system from scratch**. The end product will not use and **not include any legacy software**. The Blitz-64 project is a complete **re-design and re-write of all system software**, including assembler, linker, compiler, kernel.

My idea is this:

If an entire computer system is created by a single person, then the resulting artifact can be understood in totality by any dedicated professional.

Complexity will necessarily be contained and limited. Obviously, understanding an entire computer system is something that will be beyond the capabilities or desires of most people, but it is clear that it is significantly easier to understand a complete system than to design and implement that system. So the Blitz-64 system will, by virtue of being so limited, meet the design constraint for simplicity.

One can define a "prototype" as any design in which there are known flaws or inadequacies designed in from the beginning. A prototype is not fully functional and will fail, but it has value nevertheless. We build prototypes in order to learn.

For example, Boeing might construct a 1/10 scale prototype aircraft for use in wind tunnel testing; obviously the prototype is not intended to function as a real plane. I once built a computer out of relays; obviously this computer was fatally flawed from the moment I decided to use relays. But these prototypes are useful and we learn a lot from them.

Blitz-64 is not intended as a prototype or hobby project.

Blitz-64 is intended to be **fully functional, fully modern** computer system to address a couple of specific needs. The hardware architecture, the KPL language, and all the tools are intended to be fully functional and usable in the construction of deployable software systems. Blitz-64 is intended to be industrial-grade, power-efficient, reliable, and fast.

Is there any need for a wholly new computer system like Blitz-64?

First, a complete but non-trivial computer system would be **useful as a teaching aid** for Computer Science. In my experience, beginning programmers have a hard time getting their heads around (say) Linux. Modern systems are just too complex to grok as a whole. Successful programmers are increasingly nothing but "wizards", who learn magic phases and incantations which seem to work, without understanding why they work. This is beyond sad; it is dangerous.

Second, the market for **fault-tolerant, highly reliable systems** is expanding and, I think, **crucial for the future of civilization**. On the one hand, we have malware, hacking attempts, and hostile foreign powers trying to break or corrupt existing software. On the other hand, we are totally reliant on software for critical applications, such as **airplane avionics**, **banking infrastructure**, **medical devices**, **nuclear missile launch control**, **stock trading**, **power grid maintenance**, and **election voting**. From **space probes** to **pacemakers** to **auto braking**, we cannot tolerate a glitches or failure in these systems.

To make matters more urgent, we are moving into a dangerous political future in which cyberwar is a possibility. Terms like **cyberattack**, **cyberwar**, **cyber-terror**, **cyber-defense**, **stockpiled exploits**, and **military malware** have entered circulation. Benign errors and random failures can be bad, but targeted attacks pose a much greater threat. In a cyberattack, we might experience the simultaneous failure of multiple systems coinciding with other physical attacks. Right now, powerful nations are employing very clever guys who work hard to make existing software fail, and they do this by finding flaws and weaknesses in those millions of lines of code that nobody really understands.

To address this clear and present danger, we need a **higher level of reliability and fault-tolerance** across the board. In order to verify and certify that a system will work reliably, all parts of the system must be fully analyzed in the context in which the system will be deployed. While much of (say) Linux has been time-tested, it is clear with a system as complex as Linux, there remain holes and deficiencies. We can be certain that Linux is not bug free, yet it is widely deployed.

> *As time goes on, embedding complex legacy systems that are not truly understood by their users in safety-critical systems may become increasingly dangerous.*

The risk of systemic failures in some systems is already intolerable. The risk of concerted cyberattacks is unimaginable.

Obviously Blitz-64 will not be fully fault-tolerant and bug free from the get-go, but we do know that, other things equal, **a simpler system will be more reliable and fault-tolerant than a more complex system**.

Areas of Experimentation

The boundary between hardware and software is the **Instruction Set Architecture** (ISA). There are a several ISAs in widespread use (such as x86-64 and ARM) and there are others less widely known (such as Mips, RISC-V, AVR, SPARC, and Power).

The ISA boundary can be compared to the **border between two countries**. Generally speaking, a border is inherited from the past and its shape was determined by **historical events**. Borders are **difficult to change**. People on both sides have gotten used to the border and **people resist change**. Furthermore, a change requires agreement between two different populations with different objectives and different cultures. Agreement between groups with different mindsets is difficult.

Things change and borders are often not placed in the most practical places. The border may reflect the outcome of a forgotten battle. **Redrawing a border in a more reasonable way** may involve rivers, watersheds, and contemporary language groups. Unfortunately, a change may be impossible to achieve, even though changing the border might result in benefits to both parties.

The ISA acts as a border or interface between **two groups** of people: the **hardware engineers** on one side and the **software programmers** on the other. These groups have different educations and areas of expertise. Hardware engineers know a lot about hardware, but less about software, while the programmers know a lot about software, but less about hardware.

Programmers can make recommendations or requests to the hardware folks for **incremental changes in the ISA**, but they don't suggest sweeping changes to the ISA. Likewise, the hardware people can make incremental changes in the ISA that benefit the programmers, but are hesitant to throw out the ISA and propose something radically different.

The C language also forms a crucial landscape feature and should be considered as part of this inflexible border. Apparently, all ISA designers share one maxim:

The ISA must support existing C language programs.

Whatever changes are proposed, the ISA must continue to **support legacy software** written in C and these programs must run exactly the same as they have been running. No serious hardware designer would propose a new ISA that was unable to run C code and execute it with high efficiency. Likewise, no sane programmer would suggest that a newly proposed ISA not support C and suggest that all the low-level system software can just be thrown out and rewritten.

Another constraint that affects some ISAs is that they must continue to execute existing software. We all know that Intel has succeeded because they attached great importance to being able to **execute legacy machine code**.

Some new architectures will execute code without being recompiled. Some new architectures will execute legacy code but require all code to be recompiled. And some new architectures will execute legacy code, but require significant changes to the compiler and other systems software.

But no new ISA is rash enough to **suggest that the C language itself must be changed**. It's just too much to ask.

I love the C language. It was brilliant. C remains my language of choice for many tasks.

Nevertheless, I believe the requirement to support **C has become an intolerable constraint** on ISA design and prevents progress. Blitz-64 takes the approach that no legacy constraints, including C itself, should shape the ISA.

I am fortunate that I have done a little work on both sides of the border. My educational and professional background is clearly in software, but I have maintained an interest in electrical engineering and circuit design. I am lucky to be able to see the border from both perspectives and crazy enough to suggest something as radical as **completely redrawing this border, abandoning all legacy baggage and the restraints of compatibility with historical precedent**.

The Blitz-64 design was created in conjunction with creating the KPL language. I started by asking what I want my programming language to look like, and I worked backward, asking what I need from the ISA to execute this language.

The Blitz-64 design grew out of a similar project. The earlier project was called BLITZ, but is now called Blitz-32, if ever mentioned at all. Blitz-32 included an ISA, an assembler, emulator and compiler, an earlier version of KPL, and a small OS kernel written in KPL. I learned a lot from the years spent on that project, and Blitz-64 is my attempt to employ that wisdom to the fullest.

Anyone familiar with RISC-V will also note the influences on Blitz-64.

The Blitz-64 project is also my vehicle for experimenting and demonstrating a number of design choices that seem promising.

I will say nothing here about the choice of **Big Endian** except that I could never have gotten this to work in Little Endian. I just didn't have the extra bandwidth required to constantly mangle hex values.

I feel that **greater runtime error checking** is the right way to go, although I appreciate that there is a penalty. My approach is to assume that we must have the error checking and then ask how we can best accomplish it, with the least runtime impact. I came up with things like the CHECKA instruction (for array index checking), the Null Address Exception (for null pointer uses), the SLA (for shift-left-arithmetic), and the CHECKB/H/W instructions (for range checking).

I feel that the **plethora of integer types in C** is confusing and I feel that the problem of **integer overflow** is overlooked far too often. My solution is to fully commit to 64 bit arithmetic and to making certain that every arithmetic operation is checked for overflow. I don't trust myself to avoid overflow and have been surprised time and again that code I thought was correct overflowed. Furthermore, most of these overflows would not have been caught during debugging. We humans tend to gloss over big numbers.

For many applications, 32 bit arithmetic is dangerously limited. The numbers in programs are often big, risking overflow in almost every arithmetic operation. 64 bits numbers are really a lot bigger and accidental overflow just doesn't happen as often.

The decision to **eliminate unsigned integers** followed naturally. On reflection, I could find no good reason to include unsigned numbers, which is really the decision to exclude all negative numbers. Of course it is impossible to imagine C without unsigned numbers, but I have not yet encountered any need for them in my KPL programming. I feel that unsigned numbers cause nothing but problems.

Over the years, I have spent a lot of time debugging. **Debugging** generally consists of looking at a code and trying to figure out what it does. To get programs debugged, one needs to read and understand a program. From this, I concluded that **program readability** should be given more importance that writability.

Unix and the C language were designed for very slow computers by guys who did not grow up typing. Today, the computing environment is different: word completion, cut-and-paste, good typing skills, and slick development tools can be assumed. We can generate code much faster, but we can't read code any faster. So **KPL emphasizes readability**.

My earlier experience with grammars and parsing gave me a good appreciation for LR grammars. Like many programmers, I went through a phase of being seduced by LR parsing algorithms. But in the end, I came to understand that the syntax of a programming language is orthogonal to the expressiveness and utility of the language.

For KPL, I wanted a simple syntax and a parser that was easy to write and which could generate reasonable error messages. I came to realize that a **simple syntax is easier** for humans to read.

In the course of **simplifying the KPL grammar**, I tried to eliminate as much as possible. For example, the elimination of semicolons and mandatory parentheses (e.g., in IF and WHILE statements) reduces visual clutter. The decision to use "--" instead of "//" for comments was made because I feel that it is more effective at visually setting off comments.

When I didn't feel strongly about something, I had no hesitation in using tried-and-true ideas, such as the operator precedence from C, or the "printf" formatting string codes.

Both in the ISA and in KPL I tried to simplify and come up with a clean regular design.

Blitz-64 was not designed by a committee. There was no schedule and there were no deadlines.

I had the uncommon luxury of working at my own speed, which gave me the **freedom to ponder decisions**, when the best option was not obvious. Since there were no deadlines or milestones to be achieved, I could stop and re-work parts of the system whenever I felt that I had made a bad decision. I had the **freedom to go back and change things** that I didn't like. In fact, I look at the transition from the earlier Blitz-32 to the Blitz-64 system as going back to fix problems, although it took several years of cogitation.

I do not know how modern ISA or software design teams function, but I suspect my **total freedom from management** may not be typical.

As a result, Blitz-64 is something new, something that is quite unlike other systems.

Software Evolution

Blitz-64 proposes to create **a wholly new branch in the tree of software**. One motivation for creating derivations is to increase resilience, reliability, and fault-tolerance by increasing the number of unique alternatives, a sort of **redundancy at the level of software code base**.

When it comes to fault-tolerance, there is clearly a **benefit to diversity** in any complex system. With only one implementation, there will always be a single, critical point of vulnerability.

For example, consider the recent problems with the Boeing 737-MAX aircraft, in which a single design problem forced the grounding of all 737-MAX planes. Of course there have been costs associated with this failure, but because of the large number of different aircraft models in service, the impact on air traffic as a whole has been limited and tolerable. Because there are many competing aircraft designs, a failure in the design of one can be tolerated better than if there were only two or three different airplane models in service.

The vast majority of phones and **computers rely on a shared software base**, essentially **derived from Unix**. Imagine if all computers were to be running the exact same OS software, and there appeared **a single malware exploit** able to render that software unusable. It would be **catastrophic for civilization**. This is not hyperbole; **we have become fully dependent** on computers for money, food, power, communication, transportation. We can't tolerate serious degradation in any single one of these **life-critical systems**, let alone a combination of several.

The more we rely on a **software monoculture**, the more we risk that a single failure or successful attack may have tragic consequences. For increased fault-tolerance, civilization should explore different hardware implementations and a variety of software systems. Then, in the event of random faults or malicious attacks, some systems will be entirely unaffected and some functionality would remain intact.

There is another **benefit to variety and diversity**. Computers have effectively become ecosystems, populated by a huge diversity of interacting software components. **Software is evolving** and this evolution is heading in many different directions. We are foolish not to be creative and try some radically different approaches now and then. Even if a revolutionary approach like Blitz-64 does not lead to a revolution, such projects can still have beneficial influences on future designs.

We might even suggest that in the complex universe of software, **Natural Selection** is beginning to occur, since there is competition and some software entities (concepts, styles, code snippets, or even entire programs) survive and prosper, while other software entities are withering and fading into history.

Do we have the conditions to declare that **Darwinian Selection** is happening to software?

The analogy between the copying of code from older programs into newer programs and the **mixing of genes** that occurs in biological reproduction is obvious. **Inherited features** are being passed from older entities to newly created entities. We note that resources are limited and we observe **differential survival** among entities. **Competition** among software entities occurs whenever human users choose to download and execute one program over another. Furthermore, the **ability to survive**, flourish, and inspire new entities is **dependent on these inherited features**. Code is copied from successful apps to new apps, just as the genes of successful organisms are copied during reproduction. The Darwinian requirement of **random mutation** is crucial for Natural Selection to occur. Both ideas and code are copied from one app to another and in both cases, there are changes and often mistakes.[2] To digress into academic philosophy, I'll observe that this mutation is not fully random, so we might be observing some hybrid between Intelligent Design and Darwinian Selection.

Regardless of such philosophical musings, it's clear we need to keep creating and experimenting and not simply re-using existing intellectual property.

By creating a wholly new branch in the tree of software derivation, Blitz-64 may lead to the **exploration of new areas in software design space**, to new avenues in the **creative evolution of software**.

In any case, I hope you find Blitz-64 interesting and worth studying.

[2] If you see what I've seen some student programmers do to existing code, you'll agree that some mutations are truly random.

Document Revision

Document History

Version numbers are not used to identify revisions to this document. Instead the date and the author's name is used. The document history is:

Date	**Author**
28 October 2019	Harry H. Porter III <document created>
18 February 2020	Harry H. Porter III <document reviewed>

Permission to Copy

Please do not copy or modify this document. Any material lifted should be referenced.

About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s. The computer has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. Porter also designed and constructed the BLITZ System, a collection of software designed to support a university-level course on Operating Systems. Using the software, students implement a small, but complete, time-sliced, VM-based operating system kernel. Porter has habit of designing and implementing programming languages, the most recent being a language specifically targeted at kernel implementation.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.

Porter lives in Portland, Oregon. When not trying to figure out how his computer works, he skis, hikes, travels, and spends time with his children building things.

Professor Porter's website: www.cs.pdx.edu/~harry