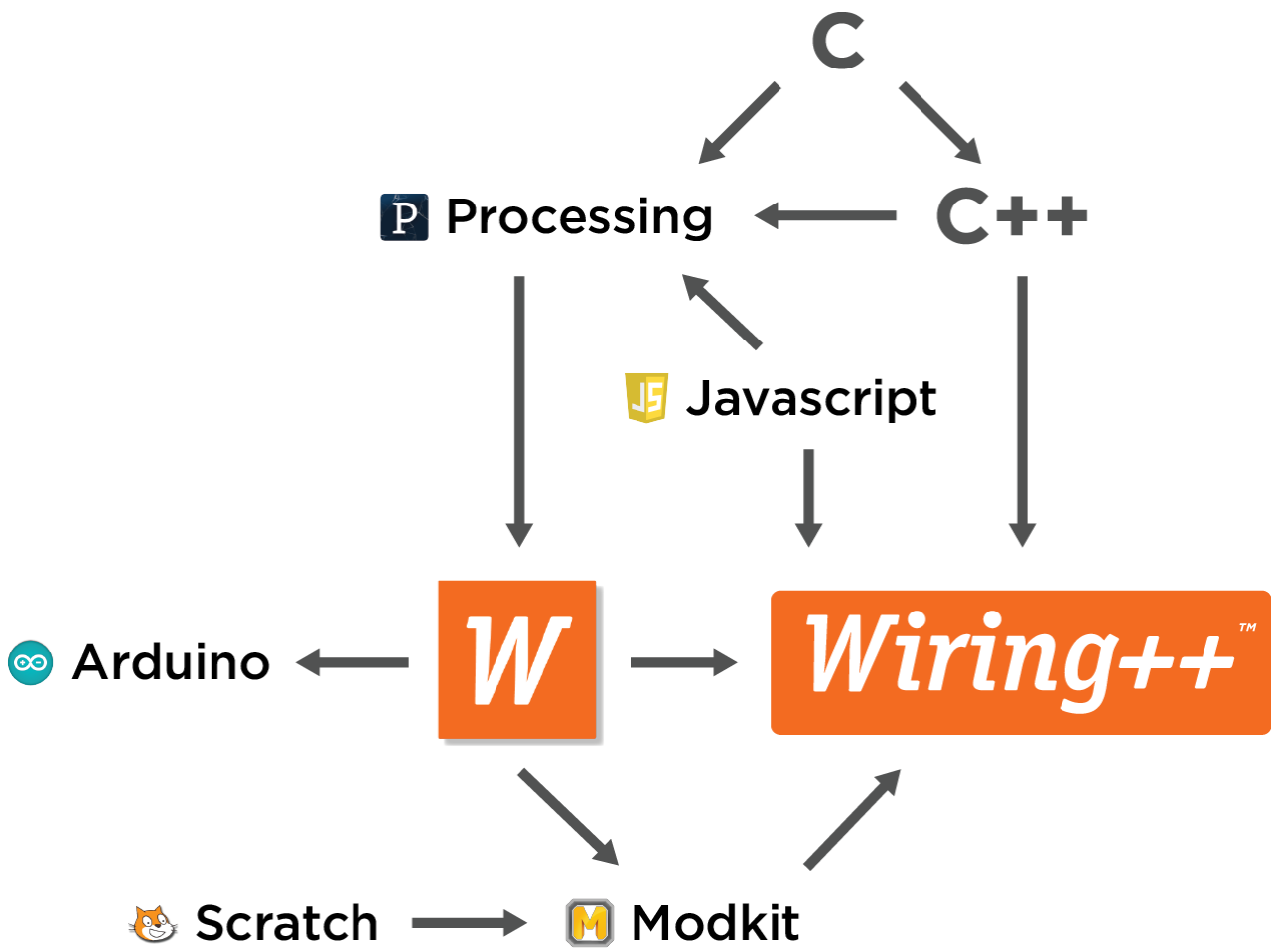
Wiring++: rationale, features and roadmap



*adapted from the Processing Language Family Tree – Casey Reas

Ed Baafi – Hernando Barragán – Alexander Brevig – Brett Hagman – Collin Reisdorf

Wiring

Wiring is an open-source programming framework for microcontrollers. Wiring enables anyone to write programs to control components and devices attached to a wide range of microcontroller platforms and to create all kinds of interactive objects, spaces and physical experiences.

Interactive Wall by Hernando Barragán and Andrés Aitken - 200X (TODO - HERNANDO - UPDATE YEAR)

The Beginning

Hernando Barragán started Wiring in the Summer of 2003 as his master's thesis project while studying at the Interaction Design Institute Ivrea (IDII). Wiring builds on Processing, an open project initiated by Casey Reas and Ben Fry, both formerly of the Aesthetics and Computation Group at the MIT Media Lab. Hernando's work was motivated by the fact that with very few exceptions, most prototyping tools for electronics and programming were targeted at engineering, robotics and other technical domains. These tools and programming languages were difficult to learn which made them unapproachable in contexts like art and design.

Hernando's original thesis included a set of interactive tables developed exclusively with Wiring, and by 2004 Wiring was introduced in the IDII classroom to teach physical interaction design. Students' work in the class was displayed as a groundbreaking exhibit titled Strangely Familiar: http://wiring.org.co/exhibition/images/book01.pdf The exhibition and overall class experience demonstrated the success of the Wiring platform as a teaching/learning/development/production tool. IDII then initiated the Instant Soup Project lead by faculty member Yaniv Steiner http://www.nastypixel.com/instantsoup/website/cover/ which developed a series of online tutorials to teach and learn through different prototyping scenarios.

A Wiring World - Wiring was already in use in dozens of Universities Worldwide by 2005

In addition to testing Wiring with student projects and in the classroom, IDII proved to be a fertile testing ground for the open source and collaborative nature of Wiring's software development. You can read more about the early Wiring journey on the Wiring homepage: http://wiki.wiring.co/wiki/About

By late 2004, Wiring had already emerged as a teaching language and electronics prototyping system. Wiring facilitated the process of learning while minimizing the struggle with electronics and programming. It also allowed designers to transfer their knowledge from platforms they already knew, like Javascript or Flash Actionscript® to the physical domain. As artists and designers were approaching technology from an aesthetic rather than a purely technical angle, Wiring enabled them to unleash their creativity which led to innovative projects and domains. This included hybridizations of high and low technologies, wearables, and adaptables (products to be adapted by its user community.)

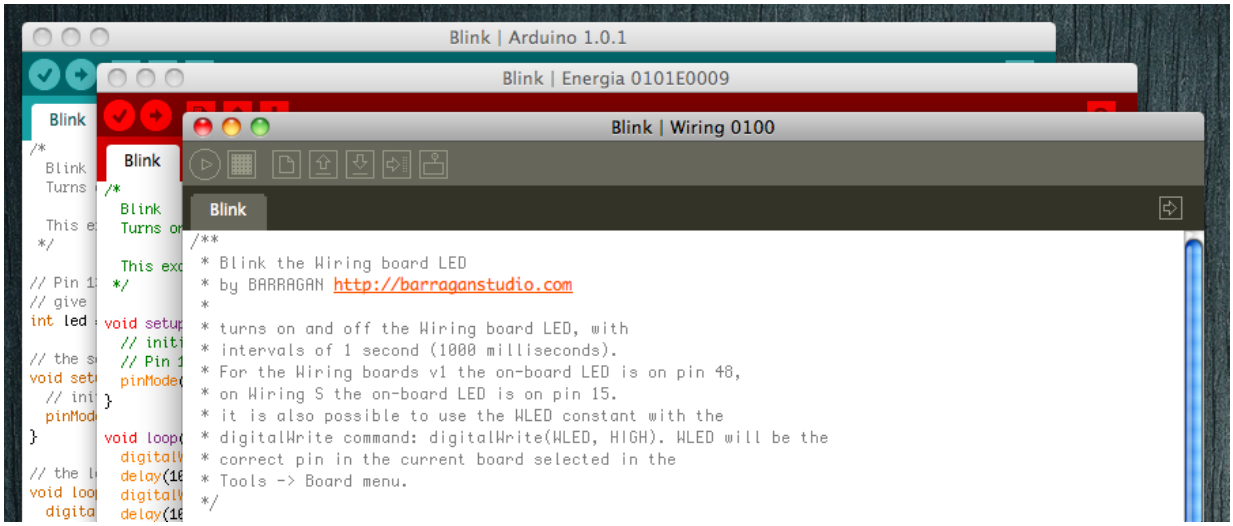
The Arduino Community

By 2005, with both the Wiring hardware and software being used successfully in dozens of Universities and educational institutions, the commercial potential for approachable electronic prototyping tools became clear to everyone close to the project. It was at this time that Wiring was forked and rebranded as Arduino.

Today, there is a large community contributing to what people collectively refer to as Arduino. As companies and individuals look to expand the reach of the open source project, they often develop new prototyping boards and software libraries to support them. While there are hundreds if not thousands of such community produced boards on the market, Arduino's

business model is to only officially support their own boards. This means that “third-party” hardware is not supported directly in the Arduino IDE, so hardware makers must provide instructions on how to “hack” support for their boards into the IDE. If the board is much different (e.g. uses a different processor family) board makers are often forced to create another fork of the Arduino environment/language which itself is a fork of Wiring. Such projects include: Chipkit (Microchip Pic32), Energia (TI MSP430 and Stellaris Cortex M4), and LeafLabs IDE (STM Cortex M4).

However, in many open source software development communities, forking is often considered a last resort when a project maintainer can not, or will not support a direction that is in the interest of a group of developers or users. Forking is often avoided, as maintaining two divergent forks causes duplicate developer effort, fragmentation of new features, and a less compelling user experience. We would argue, while the original fork of Wiring to Arduino was purely a branding strategy, that is not the case with these newer forks. Many of these groups and companies had previously approached Arduino to inquire about adding their hardware to the Arduino IDE, so these forks are necessary only because of Arduino's branding strategy, not because these developers wanted to maintain an unnecessary fork.



Arduino and Energia. Just two of over a dozen microcontroller environments that trace their roots to Wiring

Wiring Today

Brett Hagman and Alexander Brevig joined Hernando and together made Wiring what it is today. The core components and libraries were completely rewritten, integrating a schema to add new hardware and libraries. Very simple hardware definitions were created. For example, all Sparkfun hardware definitions are available as a download, and many existing hardware brands were included with the release, including bare Atmel microcontrollers. Recently, Wiring joined forces with Ed Baafi and Modkit to add revolutionary features, and direction, to the project.

The Future

After 10 years, and hundreds of thousands of users later, the future of the Wiring project lies in the strong community that has grown around its tools, whether branded as Arduino, or some other variant. While Arduino has done a great job engaging an audience by developing and distributing hardware, changes and improvements are needed in the language and the user's programming experience. We are prepared for an important turning point while preserving the original philosophy, values and ideas to mark the next stage for what has been a very successful innovation platform.

Threading and Event-driven Programming

Since Wiring was first conceived almost 10 years ago, Wiring sketches have been mostly structured within two code sections: 1) the `setup()` construct where users configure their hardware, and 2) The `loop()` construct which contains the application code which typically senses some input to control some output. As its name implies, the code within `loop` is run repeatedly and as fast as possible until the user unplugs or resets her board.

While this structure has worked well for many projects, especially simple projects with a single input sensor and simple outputs, this programming model begins to break down when adding more than one input sensor. To illustrate this, imagine a program that should do something for one second when one button is pressed and should do something else for 3 seconds when another button is pressed. Here is some sample code that accomplishes this:

```
int button1 = 11;
int button2 = 12;

void setup() {
  pinMode(button1, INPUT);
  pinMode(button2, INPUT);
}

void loop()
{
  //do some stuff if button1 pressed (brought low)
  if (digitalRead(button1) == LOW) {
    doSomething();
    delay(1000);
    stopDoingSomething();
  }

  //do some other stuff if button2 pressed (brought low)
  if (digitalRead(button2) == LOW) {
    doSomethingElse();
```

```
    delay(3000);
    stopDoingSomethingElse();
  }
}
```

This structure works fine unless you want the program to be responsive while we are still processing the first button press (i.e. while still doing something). The above code will not be able to register a second button press before `stopDoingSomething()` returns as the program is stuck in a delay. While the Wiring button library has some built-in event handlers that address this issue as well as the TimedAction library which provides another solution, a more general threading/event model would allow end-users to program their own event driven systems more easily. As we believe threading and events are essential tools when developing embedded systems, we decided to support this in the Wiring framework rather than relying on supplementary library support.

Threading and Event API

Wiring++ introduces a threading/event API that is backward compatible with Wiring/Arduino 1.x and extends the existing `setup()` and `loop()` constructs with the addition of `threads()` and `events()`. The `threads()` construct creates named threads that are callable with *`threadName`*`.start();` and the `events()` construct creates unnamed thread handlers that respond to named events e.g. `broadcast(`*`namedEvent`*`);` While this threading API provides a great deal of power, it is based on the Scratch threading/event model (scratch.mit.edu) so it is simple enough to grasp by kids and novices alike. Here is a threaded program that solves the button response problem:

```
 int button1 = 11;
int button2 = 12;

eventList{BTN_EVT1,BTN_EVT2};

void threads() {
  //watch buttons and broadcast when pressed
  thread(BUTTON_THREAD) {
    while (1) {
      if (digitalRead(button1) == LOW) {
        broadcast(BTN1_EVT);
      }
      if (digitalRead(button2) == LOW) {
        broadcast(BTN2_EVT);
      }
      delay(100);
    }
  }
```

```
}

void events() {
  // initialize stuff on startup
  when(START) {
    Serial.begin(9600);
    pinMode(button1, INPUT);
    pinMode(button2, INPUT);
    BUTTON_THREAD.start();
  }

  when(BTN_EVT1) {
    //do something when BTN_EVT1 is broadcast
    doSomething();
    delay(1000);
    stopDoingSomething();
  }

  when(BTN_EVT2) {
    //do something else when BTN_EVT2 is broadcast
    doSomethingElse();
    delay(3000);
    stopDoingSomethingElse();
  }
}
```

Let's walk through this sketch step by step -- Like any sketch, a thread-enabled sketch begins by referencing any imported libraries and creating global variables but the user will also list any named events within the eventList initializer:

```
#include "LED.h"
//.............
eventList{BTN_EVT1,BTN_EVT2};
```

Named threads are then created within the `threads()` construct using the `thread` keyword:

```
void threads() {
  thread(BUTTON_THREAD) {
    while (1) {
      //.............
      delay(100);
    }
  }
}
```

Event Handlers (i.e. unnamed threads) are created in the `events()` construct with the `when` keyword. Named events can be either events created with the eventList initializer or can be built-in events such as START which fires on startup after setup() is called (see below):

```
void events() {
  // initialize stuff on startup
  when(START) {
    //............
  }

  when(BTN_EVT1) {
    //............
  }
}
```

For backwards compatibility, the setup and loop constructs are also available. Setup is called before any user threads. Loop runs like any other thread except it is called continuously whenever its execution completes (i.e: it loops). Notice the use of the keyword `threadsafe` to indicate that the thread-safety of a given locally defined variable should be ensured by the threading implementation and will not be recreated or overwritten on every call to the thread:

```
void setup() {
   // delay startup of threads
   delay(1000);
}
void loop() {
   threadsafe int entries = 0;
   Serial.println(entries++);
}
```

Implementation Status

Wiring++ contains a preliminary threading/event implementation that is portable across 8-32 bit micros using stackless cooperative threading (protothreads). There are plans to port Wiring's threading/event API to a full RTOS (TI RTOS) on select 16 and most 32 bit micros.

Wiring++ Object-based API

From Wiring's inception, the core API was mostly a collection of C functions: `digitalWrite()`, `analogWrite()`, `digitalRead()`, `analogRead()`. As Hernando's thesis project began to be used in a number of educational institutions, different contributors with different development tools/styles began to get involved. As educators and users realized they needed certain new features, they were often added quickly to support a class or a project. While care was taken to ensure the simplicity and power of any new feature, the programming style of each addition varied. Many libraries were added with C++ style syntax (e.g `lcd.print()`)

These variations have mostly stabilized in Wiring/Arduino 1.0. Most core functionality is still C-style while most libraries feature C++ so nearly every program has to straddle between C and C++ styles. This is problematic for educators who want to focus on a certain programming style in their activities/courses and for Wiring's presentation as a cohesive domain specific language.

Complementary C/C++ Interfaces

Wiring++ addresses this by viewing the API as two complementary interfaces, one C style and the other C++ style.. Users can still mix the styles if they chose but they now gain the ability to pick one or the other style exclusively.

Wiring++ API (C++ syntax)	Wiring Classic API (C syntax)

`Serial1.println("Wiring++");`
`PIN13.pinMode(OUTPUT);`
`PIN13.digitalWrite(HIGH);`
	`println(Serial1,"Wiring++");`
	`pinMode(PIN13,OUTPUT);`
	`digitalWrite(PIN13,HIGH);`

Pins as objects

The Wiring++ API is centered around the concept of Pins as objects so rather than only being able to pass a pin to a function, Pins themselves have built-in functionality. By default, all pins on a board will be available to the user automatically through the `useAllPins()` call in the included board file. The user can just begin writing a sketch by using any pin:

```
DigitalIoPin relayPin = PIN13;
relayPin.pinMode(OUTPUT);
relayPin.digitalWrite(HIGH);
```

To greater control the resources each pin object requires or to "alias" a pin to clarify its intent within a project, the user may choose to edit the board configuration by commenting out

`useAllPins()` and employing the `usePin(PIN12);` and `usePinAs(PIN13,relayPin);` syntax.

Pin Interfaces

By moving from numeric identifiers to pins as objects, not only do we gain a new way to interact with pins, but component (library) authors can now specify what functionality a valid pin parameter will have by naming a given Pin Interface -- e.g. PWM:

```
void someComponent::someMethod(PwmPin pin, uint8_t volume){
   pin.analogWrite(volume*25);
}
```

If the user tries to pass a pin without the required functionality, a compiler error will be generated. Wiring/Arduino 1.x currently cannot detect this error and would fail silently.

The following basic Pin interfaces are available:

Pin Interface	Desctription
GPIO	Basic Digital IO
ADC	Analog to Digital converter (10 bit or greater - can add AnalogIn12 etc for more narrow interface requirements)
PWM	Pulse Width Modulation (8 bit or greater - can add PWM10 or PWM12 for more narrow interface requirements)
Serial	Simple UART (speed is limited to max speed - user must read datasheet)
SPI	Serial Peripheral Interface (speed is limited to max speed - user must read datasheet)
I2C	I2C (speed is limited to max speed - user must read datasheet)
LOW_INT	Interrupt when pin goes low
CHANGE_INT	Interrupt on pin change

RISE_INT	Interrupt on rising edge
FALL_INT	Interrupt on falling edge
please add any others	.....

Method Chaining

By introducing pins as objects we also gain the ability to add method chaining to all pin methods that would otherwise have no return value: e.g.

```
PIN13.pinMode(OUTPUT).digitalWrite(HIGH);
```

Pin Events

As Pins are central to Wiring++, it makes sense that Pins will be tightly integrated with the other framework level features such as Events. In addition to user-generated and built-in events, users will be able to write code to respond to pin-based-events that trigger automatically based on underlying Pin Interrupts or HardwareFills (see below).

For instance, a DigitalIoPin can fire a `PIN_CHANGE` event if the value of the pin changes. Users will access the `PIN_CHANGE` event through a specific Pin reference:

```
void events() {
   when(PIN13.PIN_CHANGE) {
        // do something
   }
}
```

HardwareFills

Based on the polyfill[1] concept from the HTML5/CSS3 community, HardwareFills work by introducing a software solution when low level hardware is not available. For instance, when PWM HardwareFills are enabled, any pin can have PWM functionality without the user explicitly using a softPWM library. This provides for better code compatibility and components that depend on some given functionality can be defined on more pins (e.g. simple components like LEDs that depend on PWM for fading can use virtually any pin.)

[1] http://en.wikipedia.org/wiki/Polyfill also see http://en.wikipedia.org/wiki/Shim_(computing)

Here is a list of HardwareFills and the existing libraries that would need to be ported to the HardwareFill framework:

Pin Interface	functions/methods	Existing Libraries
PWM	analogWrite/pwmWrite	softPWM
Serial	read/print/println etc	newSoftSerial
SPI	begin/end/transfer/setBitOrder/setDataMode/setSpeed/attaching and detaching interruption function etc	softSPI?
I2C	beginTransmission/endTransmission/onReceive/onRequest/read/peek/available/requestFrom/write	SoftI2C
Interrupts (LOW,CHANGE,RISING, FALLING)	not a library, external interrupts, part of the core	PinChangeInterrupts Need to be able to poll for and trigger interrupts when interrupt not available and turn pin chage into more descriptive (rising/falling etc) with additional logic.
please add any others	...	.....

Remote Resources

Another powerful concept that becomes possible with the introduction of a full object-based API is the ability to create proxies to resources like pins on remote boards. Users would be able to interact with pins from a remote board as if it were local to the sketch. Any serial communication channel could be used including a simple serial line, ZigBee, Bluetooth, etc.

```
Remote remoteBoard = Remote(Serial(Pin0, Pin1));
LED led = LED(remoteBoard.Pin13);
led.on(); //turn on LED at pin 13 on remote board
```

Implementation Details

The classic Wiring API can then be built with the Wiring++ API:

```
void digitalWrite(pin p, value v) {
   p.digitalWrite(v);
}
```

Or the Wiring++ API can be built from the classic API:

```
DigitalIoPin::digitalWrite(value v) {
    digitalWrite(digitalPin, v);
}
```

The Wiring API can remain backwards compatible with the old numeric pin id (e.g. 13) by employing a lookup table:

```
void digitalWrite(uint8_t, value v) {
   pinObjectFromNumericID(p).digitalWrite(v);
}
```

Implementation Status

Wiring++ contains a preliminary Pins as objects implementation with the C++ API based on the WIring1.0 C-style APIs.

Common Components Library

The Common Component (Abstraction) Library is a group of common components or building blocks that a user would add to her microcontroller to accomplish some task. Pretty much anything that can be added to a microcontroller pin or set of pins can be thought of as a component so it makes sense that Components are central to the next generation of the Wiring Framework. Even additional functionality that can be configured on a pin or group of pins such as Serial functionality will be organized as part of the Component Library.

Having a well defined set of building blocks will enable users (and even other libraries and components) to make use of hardware components without caring about implementation specifics. For instance, if a user buys a 433mhz button and a 433mhz communication module to connect that button to some Wiring enabled hardware, the user should be able to find and use code designed to make use of a button and should not care that it was originally created for a different button implementation (e.g. button connected to ground and a digital pin with internal pullup enabled).

Example components are:
Button, Switch, Potentiometer Knob, LED, etc

Component Events

As Components are central to Wiring++, it makes sense that Components will be tightly integrated into the other framework level features such as events. In addition to user-generated and built-in events, users will be able to write code to respond to component-based-events that trigger automatically whenever certain components are added to the system.

For instance, the Button component will automatically fire the `PRESSED` event when the button is pressed. Users will access the `PRESSED` event through a specific button reference:

```
#include "Button.h"

Button button = Button(12,BUTTON_PULLUP_INTERNAL);

void events() {

   when(button.PRESSED) {
        //do something
     }
}
```

Components and their events are listed below

Component	Events	
Button	TRIGGERED, PRESSED	
Switch	TRIGGERED, ON, OFF, CHANGED	
Potentiometer Knob	TRIGGERED,	
Potentiometer Slider		
LED		
RGB LED		
DC Motor		
Stepper Motor		
Servo Motor		
Sound Player (Speaker)		
Microphone		
Data Logger		
Timer (internal)		
EEPROM (internal)		

Common Component Interfaces

In addition to the component abstractions that allow users and libraries to swap out components based on different implementations (e.g. 433mhz button vs. a simple button) components can be more easily swapped with other components when they implement specific interfaces.

This is an extension of the pin interfaces: `analogInPin`, `DigitalIoPin`, `DigitalIoPwmPin`. Consider a “stringify” interface that declares a toString method that any component can then implement.

```
class wpp_interface_stringify {
  public:
    //toString interface (virtual function)
    virtual String toString();
};
```

Components could then inherit from the toString interface like the trivial examples below.

```
class SomeComponent: public wpp_interface_stringify {
   public:
     String toString() {
         return "Message from SomeComponent.";
     }
};

class SomeOtherComponent: public wpp_interface_stringify {
   public:
     String toString() {
         return "Message from SomeOtherComponent.";
     }
};
```

This allows users and libraries to implement functionality that only depends on a component having a particular interface rather than requiring a specific component. For example:

```
void printSomeComponent(wpp_implements_stringify& component) {
  Serial.println(component.toString());
}
```

Would print the result of toString for any component that implements it. for example:

```
SomeComponent component = SomeComponent();
SomeOtherComponent component2 = SomeOtherComponent();
printSomeComponent(component);
printSomeComponent(component2);
```

would print "*Message from SomeComponent.*" and then "*Message from SomeOtherComponent.*"

Common Component Events

In addition to component events that simplify programming and common interfaces that allow components to be swapped out with other components sharing an interface, allowing components to generate common events will also encourage code re-use and flexibility.

For instance, a button may have a `PRESSED` event but what if we wanted to write some code that could respond to any component that was “triggered”. We might think of a button press as “triggering” the button. We could think of many components that could be “triggered” such as a motion sensor so these related components could share a `TRIGGERED` event along with a Triggerable Interface. This would allow any complex component (one made up of two or more components) to accept a Triggerable component and drive an output based on a “triggered” input of any kind. This is really an extension of Interfaces: see Common Component Interfaces for more info on how Interfaces are implemented.

List of Components by Interface/Event

Below is a list of interfaces, the methods/events they support, and the components that would implement them. In time, we should have sufficient interfaces so that nearly every peripheral imaginable should be able to list in the Components column for one or more interface.

Interface	methods/events	Components
printing	print,printLine,etc	Serial,LCD,DataLogger,etc
stringify	toString	all (would return relevant info for debugging)
triggerable	TRIGGERED (event)	Button, Motion Detector, (Interrupts)
please add more..	...ideas..	...for components and Interfaces...

Component Upgrade Framework

Introduce upgrade rules at a per module level so when LED component is upgraded it contains all the rules necessary to move from one version to another. Separate components from Wiring++ spec upgrades (see backward compatibility below).

Miscellaneous Features

Lower level API building blocks

Eliminate most of the existing “CORE” libraries and make them cross platform by basing them on new core building blocks. For instance, LiquidCrystal should no longer be a core building block but can be cross platform given we build a ParallelPort building block where you can set up a virtual (HardwareFill) or actual port of arbitrary number of pins to shift out (parallelOut) parallel data. Likewise, Servo should be portable based on PWM functionality but it is currently listed as a core library probably because it doesn't use this core functionality. A little work will go a long way here.

Board/Processor definition

Board implementors should not have to write any code such as pins_arduino.h or BoardDefs.h currently required in Wiring/Arduino. Processor maintainers (eg ATMega328p or MSP430 G2553) would add code that would be organized in such a way that a simple web (JSON) format can be used to define all necessary board parameters. Not exactly thought out but should be possible. Wiring may already do this.

Deprecate #define

Avoid conflicts as in A4 that could be used for a musical note or analog pin 4. Prefer compiler errors over undefined runtime behavior when using a value that doesn't make sense for a given function/method as in digitalWrite(PIN13,A4); which is currently undefined runtime behavior in the Wiring/Arduino implementations. Haven't thought through too much but think we could use template based "define" class along with static variables to get best of both worlds. Otherwise we can choose one or other but need to improve this.

Preprocessor Framework

Separate any advanced preprocessing (beyond C preprocessor directives) from the implementation level and define the preprocessing steps in a language neutral format (JSON) to encourage many editors and build tools in the implementors language of choice.

Dynamic Resolving of Library Dependencies - Wiring++ Package Management

The current process for adding a community library is not good, at best.

We should be able to have a repository somewhere with a shared .json file. This .json file could be edited by anyone, so that anyone could add their library.

```
{
  ”name”:”SparkfunRFID”,
  ”git”:”url to git repo”, /* exactly one of these must */
  ”hg”:”url to hg repo”,      be specified*/
  ”svn”:”url to svn”
}
```

This would enable us to simply parse the user program / sketch for any exact match to the `name` of each library in that JSON. When a match is found, we could download it from the repository and prepare it for the user, if the library is downloaded we could check if an update is available and prompt the user for update approval. After the initial download, the system should check if the library has external dependencies through a file that specifies the names of the dependent libraries.

Automatic Download With Dependency Resolution and #include

An example sketch could be:

```
Serial serial = Serial(Pin0, Pin1);
RFID rfid = SparkfunRFID(serial);
LED led = LED(Pin13);

void events() {
      when (rfid.TRIGGERED) {
            if (rfid.match(RegEx(”^\\d$”))) {
                  led.blink(2,100); //authorized
            } else {
                  led.blink(1,500); //not authorized
            }
      }
}
```

So where does the RegEx come from?
Downloaded behind the scene as a dependency for RFID, because it is `wpp_interface_regexable`, and the library lists RegEx as a dependency.
At compile time (precomile), our parser would then detect the new SparkfunRFID reference, download it from ex a git repo through ”url to git repo”, resolve dependencies (and as such download RegEx if it is not already present in local libraries), add an #include “SparkfunRFID.h” to the top and initiate compile.

Backwards Compatibility

Wiring++ is almost 100% backwards compatible with Wiring/Arduino 1.x. At the structural level, users can simply (ed)...

Define a way to allow support for old versions of the framework which can provide and upgrade path from Wiring/Arduino 1.x For instance, the user would define __ARDUINO_1_0_COMPATIBILITY if she needs features from Arduino 1_0 and can't upgrade etc. This allows us to fix some of the problems Hernando has identified with the original wiring APIs such as analogWrite which could be better represented as pwmWrite.

Licensing

Wiring has traditionally had an LGPL (Lesser GNU Public License aka Library GNU Public License) license which has worked well to ensure that additions to the core are made available back to the Wiring project. As Wiring is now becoming a tool for development of real products we want to make it as easy as possible for companies to adopt Wiring into their toolkits. Currently the LGPL does present some burden to makers of commercial products who under the LGPL, would have to provide their product's object files to allow end users to re-link to updated versions of the Wiring Libraries. Furthermore, the LGPL (as associated with the GPL) is often associated with being unfriendly to business. We believe a more permissive license will better serve the future of Wiring and its users.

MIT/BSD license

Either the MIT or BSD license will be chosen for all new Wiring++ development. We will work to acquire sign-off from any legacy contributors. If legacy authors are unavailable or unwilling we must rewrite their pieces.

The following authors have been identified along with their contributions and sign-off.

Author	modules/libraries	agree to MIT/BSD?
Hernando Barragan	Core + Various Libraries/Examples, Editor sections	yes
Alexander Brevig	HAL, some various contribs	yes
Brett Hagman	Atmel AVR 8 Bit core, Tone Library, SoftPWM Library	yes

Ed Baafi	wpp_threads, wpp_core, wpp_components	yes
Paul Stoffregen http://www.pjrc.com/	WString.cpp, WString.h	yes (email consent received by Brett Hagman 2013/05/08)
please list...	most important contributors...	

Authors and Contributors

Ed Baafi – ed@modk.it
Hernando Barragán - barragan@open-work.com, b@wiring.org.co
Alexander Brevig – alexanderbrevig@gmail.com, abrevig@wiring.org.co
Brett Hagman - bhagman@roguerobotics.com, bhagman@wiring.org.co
Collin Reisdorf - collin@modk.it

*(In alphabetical order per sur name - Please add your name here if you will contribute to the spec or implementation)

Contribute!

So you're convinced that Wiring++ is the future.. How can you help with the effort? Below is a list of things you can do based on your role in the community.

Who you are	What you can do
I'm a newbie!	Start building some amazing projects with Wiring++ and make sure to tell all your friends about Wiring

I'm an Arduino user	Play around with the new APIs and let us know what you think. Make sure to educate your friends and newbies that every microcontroller board is not an “Arduino” and that if you're referring to the software environment or the domain specific language (DSL) it is more proper to refer to it as Wiring. If Wiring++ makes your life easier, start using and promoting it.
I'm a media personality, blogger, or other person of influence	Same as the Arduino user. Except make sure to tell your readers/audience not just your friends.
I'm a board maker	Get your boards supported directly in Wiring. Point your audience to Wiring instead of Arduino. Stop using Ardu- or -uino in your product name as it supports a competitor who does not support your boards. Assume your audience has never heard of Arduino. If they have they should know what a microcontroller is so talk about your product in terms of microcontrollers not Arduino. If you're also a developer, contribute to Wiring++
I'm a developer	All of the above.. But you should also be contributing to Wiring++ too. Get involved and write some code
I work for a chip maker	Talk to your manager. See if you can get 20% time to work on porting Wiring for your company's microcontrollers.

I am a tools developer (software company)	Build some amazing tools around Wiring/Wiring++

Links:
http://feed.wiring.co - Recent Wiring information.