Plumbing for the Arduino

Matthew C. Jadud
Christian L. Jacobsen
Adam T. Sampson

Revision 2011-01-24

Contents

Preface

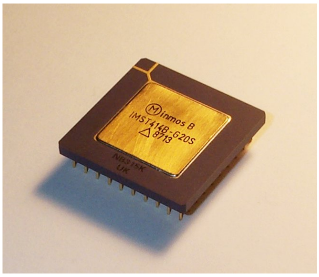
Embedded programming has always been about dealing with the real world in a timely manner.

When you push a button on your microwave, it beeps and updates the display immediately. It doesn't matter if the microwave is currently making popcorn or not—it responds in near real-time to your touch. If you've ever tried to achieve this with your Arduino (or other embedded controller), you discovered that it is very difficult to make your embedded project do two things at once—like controlling a motor while waiting for a button to be pressed. You either found yourself writing large, complex loops that constantly check everything about your system, or you found yourself reading about "interrupt vectors," and wondered if you should have paid more attention in your high school physics class.

Plumbing, and the language it is written in (occam-π), makes these problems go away.

Parallelism Yesterday

occam is an old programming language; it was developed in the early 1980's for use on the Transputer, a specialized processor developed by the British company inmos. This processor was special because it was able to switch between many thousands of parallel processes very, very quickly. It also had four special "links" that allowed it to be connected to other Transputers, instantaneously creating a distributed cluster of processors. occam made it possible, in just a few lines of code, to write programs that would run across many processors in parallel, taking advantage of these networked clusters of Transputers.



The T414.

In 2010, this may not seem impressive: for example, every computer shipped by Apple Computer has at least two cores, as is the case with many computer manufacturers today. However, we are talking about **parallel processors designed and manufactured nearly 30 years ago**. And the language, occam, has evolved—it is now called occam-π, and we have worked hard to make sure it runs on everything from your Arduino to your desktop computer, regardless of the operating system (Linux, Mac, Windows) you choose to use.

Parallelism Today

Thinking about handling things "in parallel" means handling them "at the same time." With only one processor, you can only pretend to handle two things at once—we call this **concurrency**. If you want to control a motor while waiting for a button to be pressed, or to control 64 LEDs "at the same time," you could do it with a loop. The loop would get complex, and you'd be responsible for managing all of the concurrency in your code.

Or, you could use occam-π. And on the Arduino, you could use our library of code, called Plumbing, to make these tasks much easier. For example, take a look at the code from Chapter 4: Two Blinkenlights.

```
PAR
  blink (11, 500)
  blink (12, 500)
```

With the right circuit, this code would tell our Arduino to blink two LEDs in **PAR**allel, one on pin 11, and one on pin 12. The `blink` command comes from the Plumbing library of code, and the **PAR** comes from occam-π. Combined, the language and the library of code make it easy to express ideas about problems that involve two things happening "at the same time," or (as we prefer to say), *concurrently*.

We assume that you, the reader, have little or no programming experience, but are excited to explore our tools with your trusty Arduino in hand. Please—enjoy.

The Commons

This text, as well as all of the tools you need to explore it, are free and open. Our text is made available under a Creative Commons license, our software under the LGPL, and we have chosen the Arduino (and its many variants) because of the open nature of that community as well. We encourage you to begin exploring parallel programming using Plumbing, occam-π, and your Arduino.

`http://creativecommons.org/licenses/by-sa/3.0/us/`

If you're a publisher, and are interested in working with us to produce a print edition of this text, please drop an email to `matt at concurrency dot cc`.

Bugs

If you find errors in the text, please pass them on to `bookbugs at concurrency dot cc`.

1 Getting Started

Even if you own an Arduino and have programmed it for many moons using C++ (and the Wiring libraries), we want to make sure that you're "good to go" when you start using occam-π and the Plumbing libraries.

1.1 Goals

The goals for this chapter are for you to:

1. Install the FTDI drivers for the Arduino.
2. Install our JEdit-based IDE (which will show up as the *Transterpreter*).
3. See if things work.

1.2 Installing the Drivers—or Not

This is tricky. Depending on what kind of "Arduino" you have, you may, or may not, need to install some drivers. This isn't special to occam-π or Plumbing; the same is true if you are programming your Arduino using Wiring.

Lets look at what might be the case here:

1.2.1 You have an old Arduino

If you have an older Arduino, it might have a little chip that looks like this:

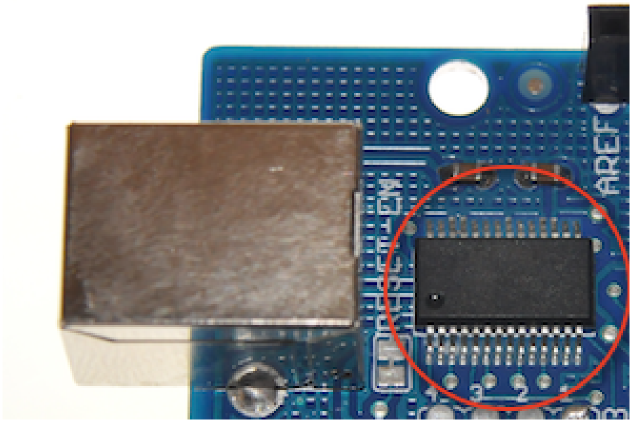


Figure 1.1: The FTDI chip on the Arduino.

If your Arduino has one of these chips, then you need to install some drivers to support the use of your Arduino. You can either use the drivers provided by the Arduino project, or you can get them directly from FTDI (`http://www.ftdichip.com/Drivers/VCP.htm`). You will want the “Virtual COM Port” drivers for your particular operating system. (These drivers are not needed under Linux.)

1.2.2 You use an adapter

You might have an "Arduino" that looks like this:

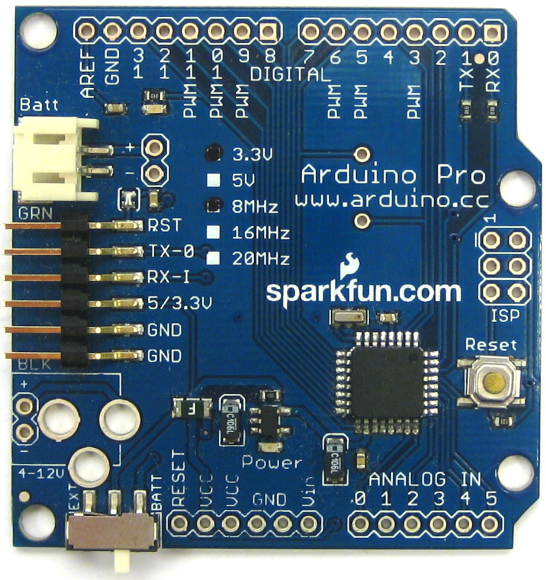


Figure 1.2: An Arduino without an FTDI chip.

Note, on the left-hand side, the six pins protruding from the board? If your Arduino requires that you use an FTDI adapter, then you will also need to install drivers. If you aren't 100% positive, an FTDI adapter tends to look something like this:

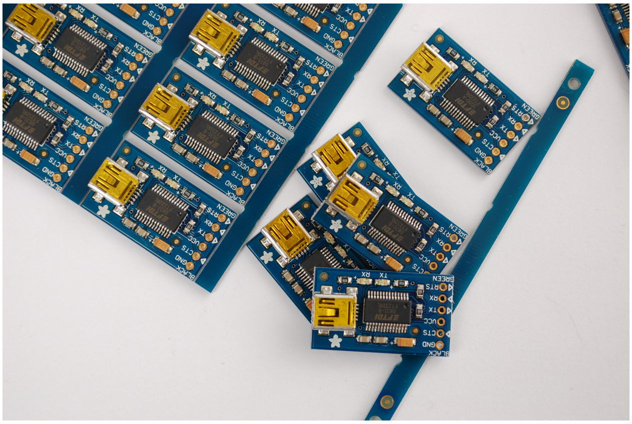


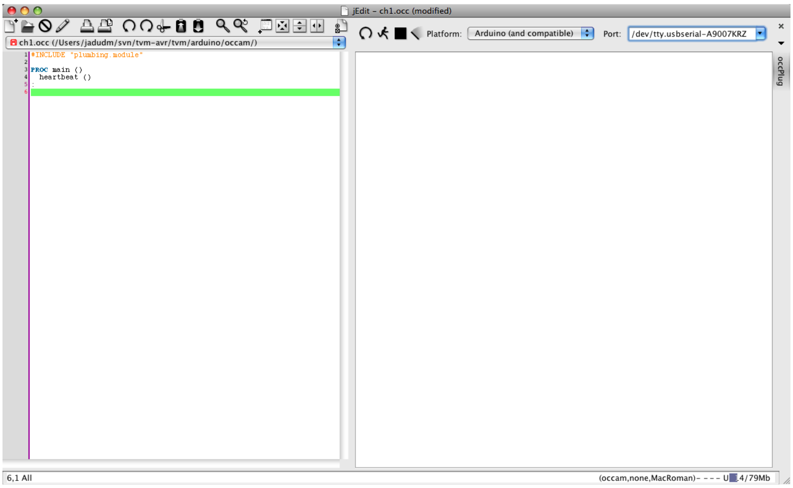
Figure 1.3: The LadyAda *FTDI Friend*, an FTDI adapter.

1.2.3 You have a new Arduino

If you have an *Arduino Uno* or an *Arduino Mega 2560* (the newest Arduinos on the block), you do not need to install anything. You're good to go!

1.3 Testing JEdit

JEdit is a free and open-source editor written in Java. It runs on Mac, Linux, and Windows. We added a "plug-in" to this project that lets JEdit talk to your Arduino. You can freely download a version of JEdit from `www.concurrency.cc` that has our plug-in pre-configured and ready to go for your choice of operating system.



ch1.occ (/Users/jadudm/svn/tvm-avr/tvm/arduino/occam/)

```
#INCLUDE "plumbing.module"

PROC main ()
  heartbeat ()
:
```

Platform: Arduino (and compatible)
Port: /dev/tty.usbserial-A9007KRZ

6,1 All
(occam,none,MacRoman)- - - - U 4/79Mb

Figure 1.4: The JEdit program editor.

1.3.1 Upload the Firmware

Before you can run occam-π programs on your Arduino, you need to upload some firmware. **Firmware** is code that lives on a processor and executes when it turns on. In the case of occam-π programs, we need to install the *Transterpreter* on your Arduino—the Transterpreter is the firmware for occam-π programs.

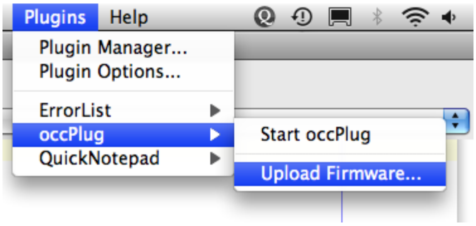


Figure 1.5: Choose "Upload Firmware" from the occPlug menu.

Selecting this menu option will pop open a window with a set of options like the following:

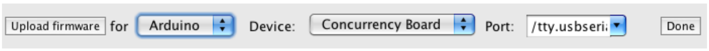


Figure 1.6: Select the correct options and upload the firwmare.

First, select which kind of Arduino you have from the drop-down menu. Then, you need to select your serial port.

- On the Mac, the name of the Arduino will look something like `/dev/tty.usbserial-A9007U6Z`.
- On Windows, it will be a COM port; you may have to type it in.
- On Linux, it should be something like `/dev/ttyUSB0`.

Once you've done that, you can click "Upload firmware," and the Transterpreter will be uploaded to your Arduino. It will then check to see everything was uploaded correctly; it should take around 10 seconds. Once it is done, you can click "Done," and begin writing occam-π programs for your Arduino.

☞ The firmware you upload is *just an overgrown Arduino sketch*. If you decide to write an Arduino program using Wiring, it will overwrite the Plumbing firmware. So, when you switch back to occam-π, the first thing you will have to do is (again) upload the firmware.

Put another way, you can freely switch back-and-forth between programs that use occam-π and C++, but you have to remember that you need to upload the firmware before an occam-π program can be executed.

2 One Blinkenlight

Because we expect you to be following along actively, we're not going to spend several chapters on theory before you get to do anything fun. Instead, we're going to get you writing code and testing it on your Arduino right away. And, we're going to start you with the simplest, and possibly most important, program you can write using Plumbing.

2.1 Goals

The goals for this chapter are for you to:

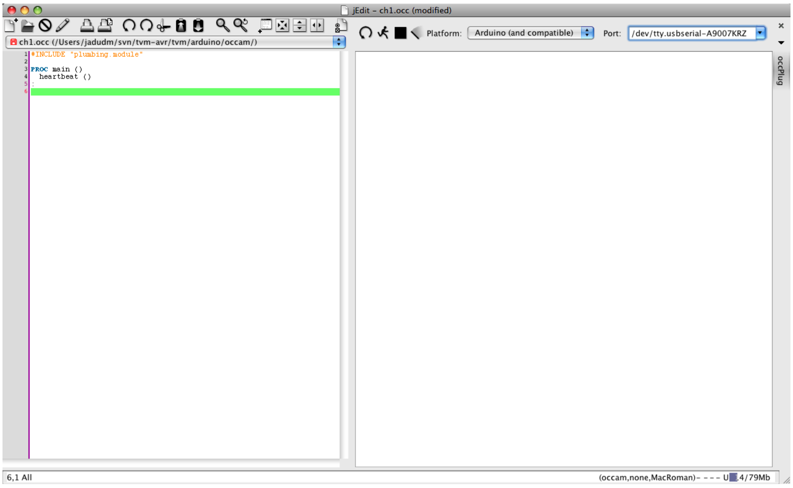
1. Write your first program using the Plumbing library.
2. Run it on your Arduino.
3. Break your first program, and fix it.

2.2 Code

In most chapters, we would now dive straight into the code you need to write or the circuit you need to build, and then discuss the patterns you should be aware of in the program you just wrote. (Programs have many patterns to them—learning to recognize these patterns is an important step in becoming comfortable with programming in any language.) In this chapter, we'll take it slow and go one step at a time.

2.2.1 Open JEdit

JEdit is a free and open-source editor written in Java. It runs on Mac, Linux, and Windows. We added a "plug-in" to this project that lets JEdit talk to your Arduino. You can freely download a version of JEdit from `www.concurrency.cc` that has our plug-in pre-configured and ready to go for your choice of operating system.


jEdit – ch1.occ (modified)
ch1.occ (/Users/jadudm/svn/tvm-avr/tvm/arduino/occam/)
occPlug
6,1 All
(occam,none,MacRoman)– – – – U 4/79Mb

Platform: Arduino (and compatible)
Port: /dev/tty.usbserial-A9007KRZ

```
#INCLUDE "plumbing.module"

PROC main ()
  heartbeat ()
:
```

Figure 2.1: The JEdit program editor.

2.2.2 Write your Program

Once you have JEdit open, you can write your first program. The first step is to get the built-in LED on your Arduino blinking—this will tell us that everything works.

```
#INCLUDE "plumbing.module"

PROC main ()
  heartbeat ()
:
```

Type the above program into JEdit. Note the first line: the `INCLUDE` line brings in all of the code that we call "Plumbing." We won't always show this line, but **you need it at the start of every one of your programs**.

Note that there are some spaces hidden there. Here's the same program, but with the spaces clearly marked:

```
#INCLUDE␣"plumbing.module"

PROC␣main␣()
␣␣heartbeat␣()
:
```

Those **spaces matter a lot**. The most important spaces are the ones on the left-hand side of each line—the **indentation**. If you get indentation wrong in occam-π, you'll get an error. We'll explore some common errors at the end of the chapter.

After copying the code, save it as `heartbeat.occ`. If you'd like to call it something else, please feel free to do so. On the Mac, you can press COMMAND-S, and under Linux and Windows, CTRL-S. Or, you can click the little floppy disk in the toolbar. Regardless of how you do it, save your work often!

2.2.3 Build Your Code

Your code is human-readable. (You may not feel that way yet, but it is.) We need to convert it from something you understand to something your Arduino understands. We would properly call this *compiling* your program. Go up to the *Plugins* menu, go down to *Plumbing*, and select the *Start occPlug* option. You'll get a new floating window that provides a few critical tools. First, you need to select "Arduino" from the drop-down menu (it defaults to "Desktop").

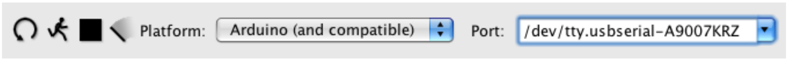
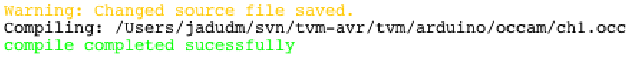

Platform: Arduino (and compatible)
Port: /dev/tty.usbserial-A9007KRZ

Figure 2.2: Compile your code before uploading.

Once you have the occPlug extension running, you can compile and run your program. When you press the round arrow on the left, our tools first check to see if your code is grammatically correct, and then transform your code into something that will run on the Arduino. If you made any mistakes in typing in your program, this is where you'll get one or more seemingly incomprehensible errors. Think of them not as "errors" but instead as "learning opportunities."

If you code compiled, you'll see a message that looks something like this:



```
Warning: Changed source file saved.
Compiling: /Users/jadudm/svn/tvm-avr/tvm/arduino/occam/ch1.occ
compile completed sucessfully
```

Figure 2.3: Compile your code before uploading.

If you get the "green light," so to speak, you're good to go! Hit the running dude, and your code will be sent to your Arduino and begin executing.

2.3 Patterns

When you're learning to program, it is important to see the patterns that exist in the code. Sometimes these patterns are strict rules that you cannot violate, or you will encounter an error. We will also see patterns that represent how programmers typically do things. What follows is the first of these strict rules—which we call **syntax**—that you will encounter in occam-π programs.

2.3.1 The PROCedure Definition

The first pattern you see in this program is the definition of the procedure called `main`. Figure 2.4 on the following page) removes the details of the code so we can focus in on that pattern itself.

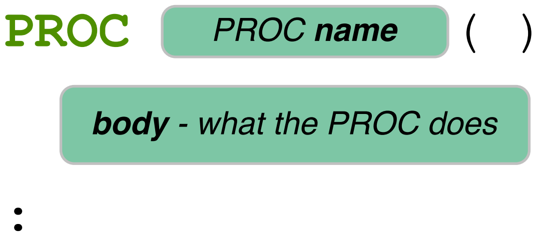


Figure 2.4: A procedure definition.

You will see a lot of **PROC**edure definitions while you are using Plumbing, because they help us keep our code organized. There are a few things we can say about every **PROC** definitions that you will encounter:

- A **PROC**edure definition always starts with the word **PROC**.
- **PROC** is always followed by the name of the procedure; in our first program, the procedure is called `main`.
- A set of parentheses follows the name of the **PROC**; then we hit return. (We'll learn where and when to put things inside those parentheses later!)
- We will call the stuff inside the **PROC**edure its **body**. This is the code that makes up the actions of the **PROC** we are writing. In our first program, there is only one line of code in the body of `main`.
- The **PROC** ends with a colon, all by itself on a line. This

signals that we're done defining this particular **PROC**, and are ready to start writing another.

While it may seem daunting to have all the rules spelled out that way, we're trying to be clear and help you see that programming is not so much a mystery as it is a matter of following rules. Every program you write will end with a **PROC** that out of habit you might call `main`. That is because the **PROC** at the end of your program is the first one that will be run by your Arduino.

2.3.2 A PROCedure Call

Look again at line 4 of our first program:

```
1  #INCLUDE "plumbing.module"
2
3  PROC main ()
4    heartbeat ()
5  :
```

From what we have just learned about **PROC** definitions, we can say that the body of the **PROC** named `main` is one line long. That line is a **PROC**edure call. A procedure call typically looks like Figure 2.5 on the next page.

A procedure call is a way of saying "someone else wrote a bunch of great code, and I'd like to use it here, please." In this case, we wrote a **PROC** called `heartbeat`, and we're making it available for you to use. That code blinks the LED on the

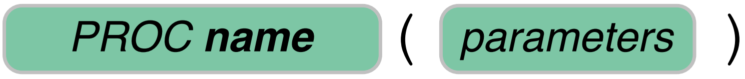


Figure 2.5: A procedure call.

Arduino.[1] We would say that the **PROC** called `heartbeat` is part of the Plumbing library. Or, if you prefer, when you are programming using Plumbing, `heartbeat` is one of the procedures provided for you to use.

2.4 Breakage

With every new piece of code or pattern, there are a dozen ways to break it. Because we want you to be *empowered explorers*, we're going to help you break your code at the end of every chapter. We highly recommend you experiment not only with writing programs that work, but also with writing programs that **do not** work. You should keep a notebook of all the errors you encounter while learning to program; while it may seem easy to fix them now, you may become confused when you decide to tackle a program of your own design, later. Many who study programming are content for it to seem magical, instead of tackling it systematically. A detailed

[1] You will learn how to write everything we show you in the first ten chapters, so you'll get to see all of the magic soon enough.

notebook goes a long way towards helping dispel mystery.

Further, it is important to note that we are encouraging you to explore common errors made by many people learning to program in all languages. As it happens, we're pretty comfortable programming in occam-π—which is why we know about these errors. Put simply, **these are mistakes we still make**. I guess we're trying to say that there is no shame in these kinds of programming errors... and our hope is that, by exposing you to these errors directly, it will help reduce some of the frustration that sometimes accompanies learning new things. Go to it!

Misspellings

One of the most common errors made by beginning programmers in any language are typos and misspellings. What happens if you type `PRC` instead of `PROC`? `maim` instead of `main`? `hearbeat` instead of `heartbeat`?

Capitalization

What happens if you write `proc` instead of `PROC`? Likewise, `Heartbeat` instead of `heartbeat`?

Forget the colon

What happens when you leave the colon off the end of a **`PROC`** definition? This is a common error.

Forget the parens, I

What happens when you leave one or both of the parentheses off the **`PROC`** definition?

Forget the parens, II

What happens when you leave one or both of the parentheses off the **`PROC`** call in the body?

Indentation

What happens if you indent the body of the **`PROC`** by one space instead of two? Three spaces?

2.4.1 Programming Strategies

Remember, learning to program in any language can be a frustrating experience. Here are a few tools you can use to help clear the hurdles you might encounter:

Community
We have a website with more information and mailing lists you can join; take a look at `http://concurrency.cc/`. If you get stuck, join the discussion list and ask a question. We're there to help.

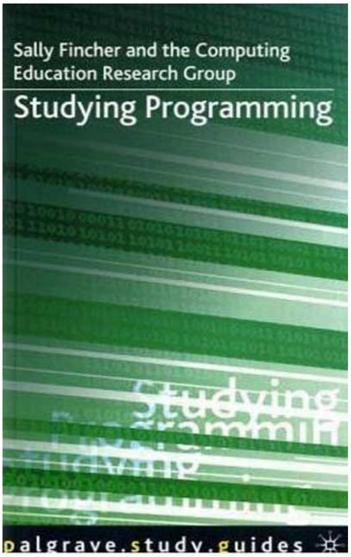
Attention to Detail
Spaces matter in Plumbing, and they are *invisible*. Be careful about what you do, and begin learning to look with the eyes of a programmer: start looking for patterns and the invisible parts of your code.

Take Notes
As you discover new kinds of mistakes you've made, take the time to make note of them, as well as your analysis of how you fixed them. Eventually, you won't need to make the notes, because you'll make fewer mistakes.

Take a Walk
...or a roll, or whatever. You should especially do this if the weather outside is your particular favorite. Let your mind wander as you wander the outdoors.

Take a Shower
Or do whatever relaxes you and breaks your routine. Perhaps you prefer a bubble bath? Either way, don't forget your rubber duckie.

2.5 Other Resources



There is one last resource we'd like to recommend. *Studying Programming* by Sally Fincher and the Computer Science Education Research Group at the University of Kent in Canterbury, England, is a wonderful resource. It was written as a study guide for someone attempting to learn to program. Some of the hints and strategies we will be presenting throughout this text are informed by that text—but there is much more there than we can include here. We should know—Matt was one of the co-authors of both texts.

While you might consider it a conflict of interest, we'd rather consider it expert opinion. There are few (if any) other texts that provide guidance and strategy for the novice programmer. Give it a look. And remember, Matt isn't making any money on either of these books, so this is just the best resource recommendation we can make to new programmers.

3 Speedy Blinkenlight

By changing just one line of code, you can control the speed at which the LED on your Arduino blinks.

3.1 Goals

This chapter just takes a small step from Chapter 2.

1. Build our first circuit.
2. See how digital voltages are either **`HIGH`** or **`LOW`**.
3. Learn about the `blink` procedure in Plumbing.
4. Learn how to control the behavior of a procedure by changing its parameters.

3.2 Building the Circuit

We could just blink the built-in LED, but we need to know how to connect multiple LEDs to the Arduino if we're going to work through Chapter 4: Two Blinkenlights, where we'll learn to blink both the built-in and external LEDs at the same time.

Your Arduino will live at the center of a number of increasingly complex circuits. We call them *circuits* because they are a loop that makes an electrical connection from a voltage source (in this case, pin `13`), through one or more electronic components back to "ground" (typically labeled `GND`). Our goal is a circuit that looks like Figure 3.1.

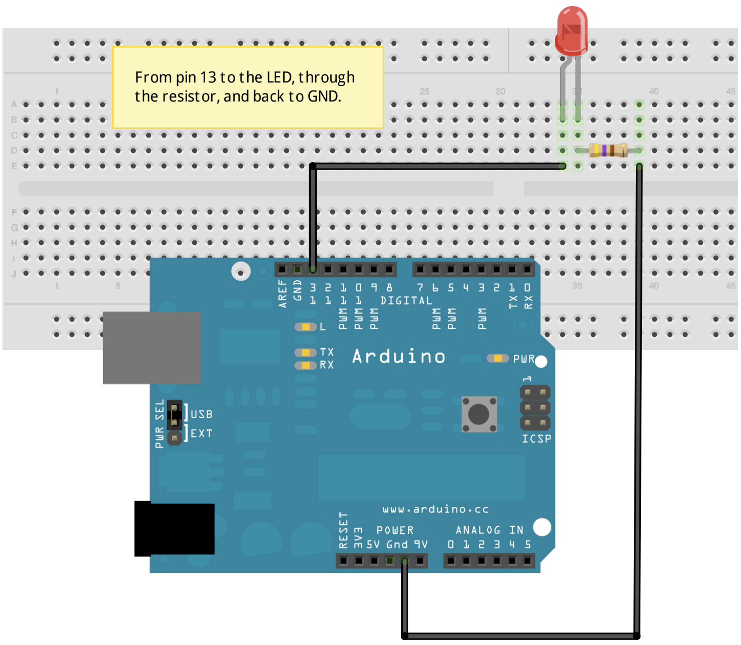
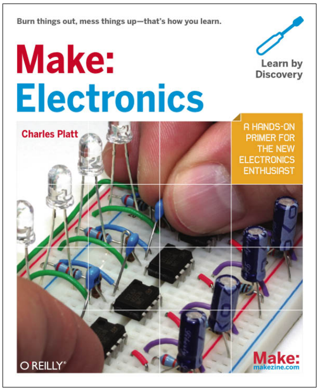


Figure 3.1: Our target circuit.

We'll build the circuit up one step at a time. What we won't be doing is teaching you electronics—for that, we recommend you pick up a copy of *Make: Electronics*[1], or for a more in-depth treatment, perhaps a used copy of *The Art of Electronics*[2] by Horowitz and Hill. Because this is our first circuit, we'll take a bit more time, but in future chapters, we'll be assuming that you have a resource like *Make: Electronics* available to you. This is more a book about programming with Plumbing than a book about the fundamentals of electronic circuit design.


Burn things out, mess things up—that's how you learn.
Make: Electronics
Learn by Discovery
Charles Platt
A HANDS-ON PRIMER FOR THE NEW ELECTRONICS ENTHUSIAST
O'REILLY
Make: makezine.com

A good resource.

[1]See `http://oreilly.com/catalog/9780596153755/` for more information about *Make: Electronics*.

[2]See `http://frank.harvard.edu/aoe/` for more information about *The Art of Electronics*.

3.2.1 The Breadboard

The breadboard provides a foundation for building and testing small circuits. Breadboards come in many shapes and sizes; if you have a small Arduino kit, you might have a breadboard like that pictured in Figure 3.3. Note that things in a column (on one side of the gutter) are connected, but things on opposite sides of the gutter are not.

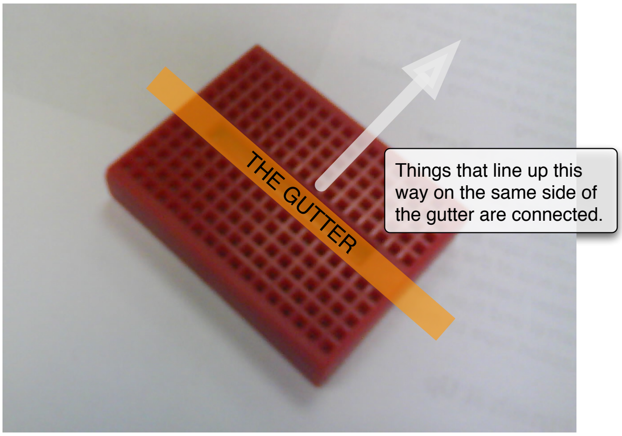

THE GUTTER
Things that line up this way on the same side of the gutter are connected.

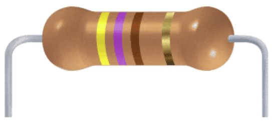
Figure 3.3: How a breadboard works.

3.2.2 The Arduino

With the Arduino turned off (unplugged from the USB port), take a wire and connect it from pin `13` to one of the columns in the breadboard. Perhaps start with the left-most column, and we'll build our circuit from left-to-right. (See 3.1 on page 31 if you get confused—it's rather accurate.)

3.2.3 The Resistor

If you plug your LED directly into an electronic device—even one as small as the Arduino—you might burn it out. If you manage this, it will probably flash brightly once and never light again. Therefore, we need a resistor to help limit the flow of current through our circuit so the LED doesn't get fried. If you need a rule of thumb (which is not always correct!), a 1KΩ resistor is typically more than enough to protect your LED.[3]


A 470Ω resistor.

Resistors are the little barrel-shaped things with different colored stripes on them. Those stripes tell you what resistance value they have.[4] Plug one end of the 470Ω resistor (🟨🟪🟫🟨, Yellow Purple Brown Gold) into the same column of the breadboard as the wire you connected from the Arduino, and the other into an unused column further to the right.

[3]If you want something better than a guideline, look up Ohm's Law on the Wikipedia: `http://en.wikipedia.org/wiki/Ohms_law`.

[4]See `http://en.wikipedia.org/wiki/Electronic_color_code` for more information about the color codes on resistors.

3.2.4 The LED

LEDs are a kind of diode. A diode is a device that only allows electric current to flow in one direction. Therefore, if you connect an LED up "backwards," nothing will happen. (This is true up to a point—enough current will destroy an LED even in a backwards configuration.)

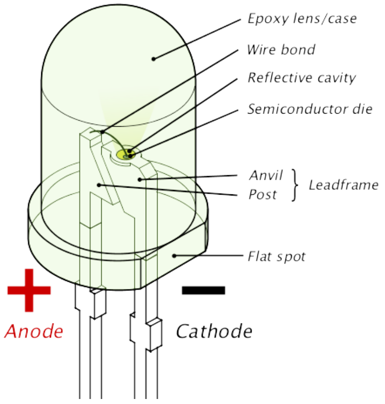


Figure 3.5: The internals of an LED.

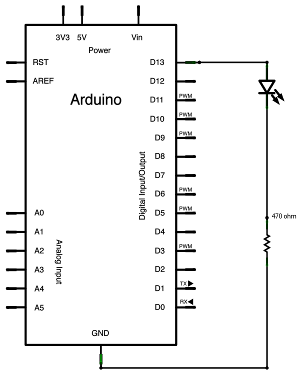
Some people say "the long leg of the LED is the negative leg." This is true, but if your LED gets mangled, it becomes difficult to tell which leg is which. Instead, look at Figure **??** on page ??, and find the "anvil." The anvil is the larger of the two bits inside the LED, and it is *always* the negative side of the LED, meaning the "post" is *always* the positive side.[5]

[5]See `http://en.wikipedia.org/wiki/Led` for more information.

Plug your LED into the breadboard so that the positive side is in the same column as your jumper wire from `13`, and the negative side is in a column with one end of your resistor.

3.2.5 Completing the circuit

Lastly, to complete the circuit, connect the negative side of the resistor to the `GND` pin on your Arduino with a jumper wire. You should now have a complete circuit that looks like Figure 3.1 on page 31. To the right is the equivalent circuit diagram that you might find in a text on electronics; you can see the source of the current (pin `13`), the LED (the triangle with the arrows coming off of it), the resistor (a squiggly line), and a connection back to ground (GND on the Arduino).



A schematic of our circuit.

Now, you should be able to plug in the USB cable, and type in the code from this chapter.

3.3 Code

```
1 PROC main ()
2   blink (13, 500)
3 :
```

Figure 3.1: The `blink` procedure lets you control how rapidly an LED blinks and which pin the LED is connected to.

3.4 Patterns

In the previous chapter, we saw the `heartbeat` procedure. In this chapter, we are introducing a new procedure called `blink`. Unlike `heartbeat`, `blink` lets us control both which LED we are blinking as well as the speed at which the LED blinks.

On line 2, we can see that a **PROC**edure called `blink` is being called. This is just like `heartbeat`—the code for that procedure is provided by the Plumbing environment. Note again the indentation—like `heartbeat` (from Chapter 2), `blink` is indented by two spaces. However, instead of an empty set of parentheses (as was the case with `heartbeat`), there is stuff in-between them. The numbers (**`13`** and **`500`**) are the **parameters** of the procedure `blink`.

```
blink (13, 500)
```

Multiple parameters are always separated by a comma.

Parameters are values that we give to procedures that let them do different things based on the values we provide. For example, the first parameter to the procedure `blink` is the number **13**. This tells the Arduino which bit it should be turning on and off. Technically, we would say that the pin to which the LED is attached is being driven **HIGH** and **LOW**. As we explore more of the basics of electronics, you'll come to understand why we say **HIGH** and **LOW** instead of "on" and "off."

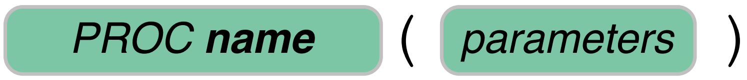


Figure 3.7: Parameters go inside the parentheses.

The second parameter to `blink` is the amount of time that we want to go by between when the LED is turned on and off. You might think that **500** is a rather large amount of time—until you realize that it is a value in *milliseconds*. The prefix *milli* means *one thousandth*. 1000 milliseconds (or 1000ms) equals 1 second. Therefore, half of a second is 500ms, and a tenth of a second is 100ms.

3.5 Experimenting with Changes

You can experiment with a few things at this point. For example, you could connect your LED up to pin 12 instead of 13. After changing the circuit, you would then need to modify your code.

```
PROC main ()
  blink (12, 500)
:
```

Figure 3.2: Blinking an external LED on pin 12.

The first parameter to `blink` tells the procedure which pin it should be driving **HIGH** and **LOW**. If we connect the LED to pin 12 on the Arduino, we need to update the procedure call. Likewise, we can change the rate at which the LED blinks. Currently we are using the value 500. What happens if you make it higher? Lower?

3.6 Breakage

There are a number of things that can break in this chapter.

3.6.1 Break your circuit

You've completed your first circuit. However, you might have done something wrong, in which case, nothing will work.

You can intentionally break a few things without damaging your Arduino or burning out your LED.

Flip the LED
If you flip the LED around, it won't light. In fact, it might burn out. Up to you if you want to test this.

Try another resistor
Try an 10kΩ resistor instead of a 470Ω resistor. You could try smaller resistors... but again, the LED might burn out. Up to you. (Trying larger values is safe.)

Wire wiggles
If you wiggle a wire out of place, you'll break the circuit. Then, nothing will work.

You can safely do each of these things, and see how your circuit fails to blink properly.

3.6.2 Break your program

There are quite a few ways you can break the software in this chapter—even though it is only three lines long!

Wrong pin
If you forget to change the pin number from `13` to `12`, then you'll blink the wrong LED. Or, for that matter, if you blink pin `11`, nothing will appear to happen at all.

Crazy parameters
Try replacing the number `12` with `TWELVE`. See what happens.

Crazy parameters II

Try replacing the number `12` with `122`. See what happens.

Too many parameters

Try using a parameter list like `13, 12, 500`. That is, your code would look like:

```
blink (13, 12, 500)
```

Parameters too big

The parameter for the speed of the LED blink is an integer—a whole number without any decimal parts, if you prefer. Computers can only keep track of numbers that are so big (or small). How big can you make the blink speed?

Blink too fast

What happens if you make the blink speed too small (e.g. zero)?

Fractional blinking

What happens if you try and blink the LED every 100.5ms?

Remember, keeping track of the mistakes you make helps you know how to deal with them when you encounter them in your own programs later.

4 Two Blinkenlights

So far, we have not done anything with Plumbing that you could not do in any other language. In this chapter, we will show you how you can use Plumbing to blink two LEDs at different speeds in *just four lines of code*.

4.1 Goals

1. Blink two LEDs together or separately.
2. Learn about **`PAR`**, the occam-π construct for building **`PAR`**allel programs.

4.2 Build the circuit

You may have modified things as part of your explorations in the last chapter. For Chapter 4, you'll need two LEDs connected to your Arduino—one connected to pin `11` and one connected to pin `12`. The cathode of each LED should be connected to ground through a resistor with a value between 470Ω and 1kΩ. Figure 4.1 on the next page shows how you should configure your Arduino.

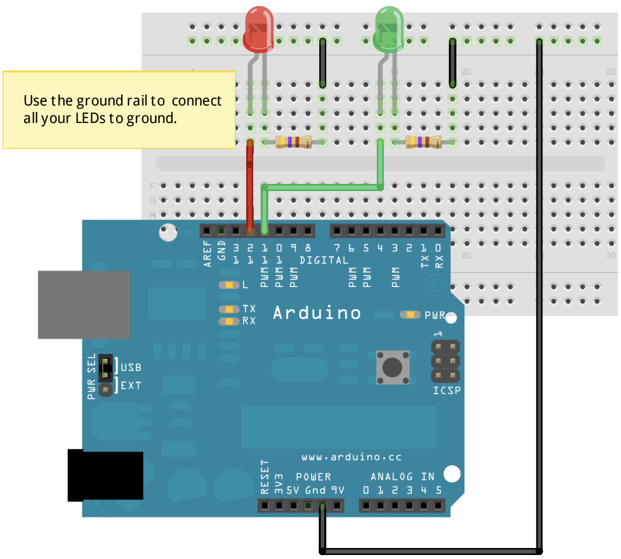


Figure 4.1: A circuit connecting two LEDs to the Arduino.

In the previous chapter, we used `blink` to turn one LED on and off at a rate of our choosing. Now, we don't want to blink just one LED, but instead want to blink two. Unlike many programming languages, occam-π gives us a way of saying this directly. We can write a program that says "please blink the LED on pin **11** at the same time as you blink the LED on pin **12**." There is no other language available for the Arduino that lets you express this so simply.

4.3 Code

```
PROC main ()
  PAR
    blink (12, 500)
    blink (11, 500)
:
```

Figure 4.1: We can `blink` in **PAR**allel.

4.4 The PAR pattern

In the previous chapters, we've written a `main` procedure that only did one thing. Many interesting programs need to do lots of things *at the same time*. If we want two things to happen at the same time, we use **PAR**.

So far, we have seen that a **PROC** may only contain one process, and that is indented by two spaces. To do two things simultaneously, then we need to use something like a **PAR**. The **PAR** itself is indented two spaces, and then everything underneath it is indented two more spaces. We can put *any number of additional processes* underneath a **PAR**, and occam-π will take care of running all of them in parallel. In the code in Listing 4.1, you can see that we are asking Plumbing to run two things in parallel, because there are two procedures underneath the **PAR**. This pattern is illustrated in Figure 4.2 on the next page.

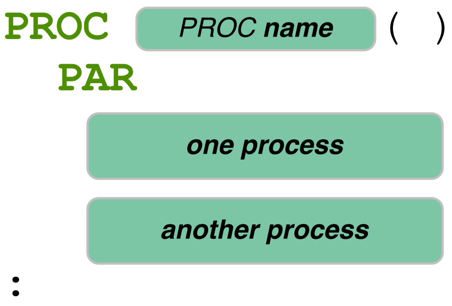


Figure 4.2: Executing two processes in parallel.

As it happens, we are asking Plumbing to run two `blink` processes. Our program asks it to blink the LED on pin **12** at the same time as we blink the LED on pin **11**. Type in this new program and upload it to your Arduino; if all goes well, both LEDs should be blinking in synchrony.

4.4.1 The truth about PAR

We have been saying that when you indent processes underneath a **PAR** that they will happen "at the same time." This is what it means when we say two things happen "in parallel." However, your **Arduino only has one processor**. It is typically the case that a device with only one processor can only do only do one thing at a time. Despite this, we are clearly executing a program that blinks two LEDs at the same time!

Figure 4.3: Parallel processes are juggled on the Arduino.

This may seem like an odd state of affairs: we wrote a program that blinks two LEDs in **PAR**allel, but the Arduino can only do one thing at a time. While occam-π was originally designed so you could write programs that run on many processors, it can also be made to work just fine on a single processor. To make this possible, we wrote a piece of software called the Transterpreter[1] that runs occam-π programs and provides the illusion of parallelism by juggling all your parallel processes around, making sure each one gets a turn.

The illusion of parallelism is called **concurrency**. This should, we hope, provide a clue as to why our website's name is `concurrency.cc`!

[1] `http://www.transterpreter.org/`

4.4.2 Explorations

It is likely that, at this point, you are chomping at the bit to do much more... you're now thinking about running motors, and sensors, and all kinds of things in parallel, doing things were never knew how to do before. For the moment, we're going to continue to explore occam-π and the Plumbing library one step at a time.

Based on what we've done so far, you should be able to do some additional explorations on your own.

Vary the parameters
: Currently, both LEDs are blinking at the exact same rate. Try changing the rate at which they blink by varying the second parameter to the `blink` process.

Add more LEDs
: If you have more LEDs, you should be able to wire them up like your first LED using more pins on the Arduino. Then, add more `blink` processes underneath the **`PAR`** that will turn that pin on and off.

And a bit of science...

If you have an oscilloscope, you can do some testing on your Arduino. How fast, for example, can you `blink` an LED? Does `blink(13, 1)` really turn an LED on and off at a rate of 1ms? Or, is it slower than that? Record these numbers in your notebook, and see if you can work out the limit as to how quickly you can drive an LED on and off when using Plumbing.

4.5 Breakage

There are a lot of neat ways to break the code in this chapter.

Indentation of `PAR`

What happens if you fail to indent `PAR`, but indent all of the `blink` procedures? We make this mistake in our code all the time.

Lowercase `PAR`

What happens if you make **par** all lowercase? What if you only capitalize the **P**?

Indentation of `blink`

Try indenting each `blink` process four spaces instead of two. What happens?

Multiple `blink`s on one pin

Modify your program so that two of your `blink` processes refer to the same pin number. (Note this breaks *after* you upload your program, not before!)

Replace one `blink` with a `heartbeat`

Modify your program so the `PAR` looks like this:

```
PAR
  heartbeat ()
  blink (11, 500)
```

What happens? Does this break anything? What if you `blink` pin `13` instead of `11`?

Two `heartbeat` processes

What happens if you run two `heartbeat` processes in parallel?

Waiting for the World

At this point, we have learned a few things about both electronics and programming. Learning that we can write code that expresses ideas regarding parallelism with **PAR** is, we think, a fundamental and world-changing concept for many programmers. If it felt "natural," that's a good thing.

Waiting.

In terms of electronics, we learned how to connect an LED to our Arduino. In terms of programming, we learned some of the basics of the syntax of occam-π. Next, we're going to learn a bit about *waiting* and *signaling*. Like everything else about Plumbing, we'll do this in **PAR**allel—one process will be responsible for signaling that something has changed (a button is pressed, for example), while another waits to see what has happened.

5 Push the Button

Writing programs that control lights and motors is fun, but to build really interesting creations we need to be able to respond to events in the world. In the next few chapters, we'll learn how to respond to events that take place in the world around us.

5.1 Goals

Connect a button to the Arduino and use it to turn an LED on and off.

5.2 The Circuit

For this circuit, we'll add a button to the Arduino that lets us turn an LED on and off. To do this, we'll need to see how to attach a button to the Arduino.

You will need:

1. Your Arduino
2. A button

3. A 10kΩ resistor
4. Some jumper wire
5. An LED and a 470Ω resistor

A picture of what you're going to build (and the circuit diagram) can be found in Figure 5.1.

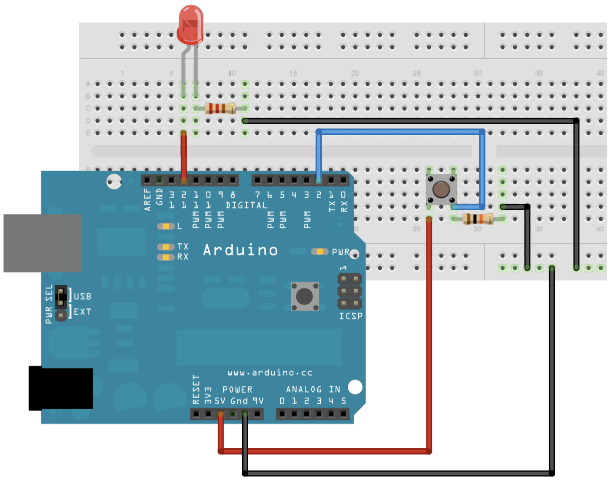
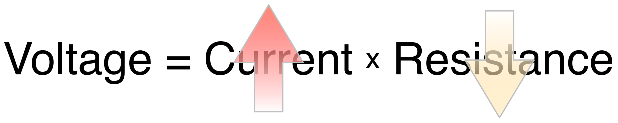


Figure 5.1: Connecting a button to the Arduino.

The newest and most interesting part of this circuit is the addition of the button. One leg of the button will be connected to the +5V pin on the Arduino. The opposite leg will be connected to ground through a 10kΩ resistor (🟫⬛🟧🟨, Brown Black Orange Gold). From the same column as the resistor we will connect a wire to pin 2 on the Arduino; we will use pin 2 to detect whether the button has been pressed.

Getting this circuit wrong could have some unpleasant consequences for your Arduino. Remember, there are plenty of texts that will teach you more about electronics than this one. That said, you should know why that 10kΩ resistor is so important. If you left it out, pushing the button would be equivalent to connecting `+5V` directly to pin 2. Why is this bad? The only resistance between the voltage source and pin 2 would be the resistance of the wire; it turns out that this is a very small number. If we look at Ohm's Law[1], we see why this is bad.



If the resistance is very small, and the voltage is constant (`+5V`), then the current must be big. Resistors limit the current flow in a circuit; without it, we would sink more current into pin 2 than the Arduino can handle, frying our processor. By including the resistor—specifically, a pretty big one—the current drops a great deal, and pressing the button does not mean death for pin 2.

[1]FIXME Wikipedia link

5.3 Pictures and Code

Programming is an incredibly visual activity. If you observe two experienced programmers discussing a programming problem, you'll discover that they make heavy use of diagrams in their conversation. When we're working with occam-π, we also make heavy use of diagrams. This is especially nice because we can translate those diagrams directly into code.

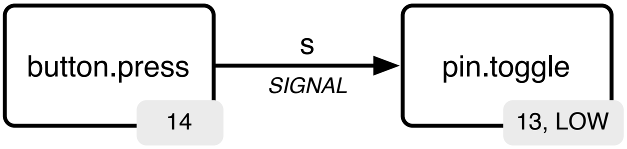


Figure 5.2: A communicating two-process network.

We call this a **process network**. Each large box represents a **process**, which is a piece of code that is churns away independently of every other process. We use small, gray boxes to indicate **parameters** that adjust the alter the behavior of the process they are attached to. So, in Figure 5.2, we would say that `button.press` has one parameter, which is **2**.

The arrow between the two processes is a **CHAN**nel. Specifically, it is a communications channel—a wire—that connects one process to another. When we look at Figure 5.2, we can tell that the process `button.press` communicates over a channel called `s` with a process called `pin.toggle`. We know that `pin.toggle` does *not* talk with `button.press`, simply because the arrow tells us which way the communications take place.

5.3.1 From pictures to code

We can, by following a few simple rules, convert the diagram on page 54 into code. **All the information we need to write a valid occam-π program is contained in the process network diagram.** It is a lot more fun to design programs when you can do it visually in a sketchbook as opposed to in front of a screen.

First, we see there are two **processes**. We can write down their names.

```
button.press ()
pin.toggle ()
```

Next, we know that both processes have **parameters** associated with them (the gray boxes). We're going to need to include those parameters in our code.

```
button.press (2)
pin.toggle (12, LOW)
```

Because there is an arrow connecting these two processes, we're going to need a CHANnel to communicate over. But here's the trick: **all communications must happen in PARallel**. Why? Can you have a phone conversation with someone where you speak and *then* they listen? Or, do they have to be listening *at the same time* as you are speaking? See.

Communications must happen in PARallel.

To run our two processes in parallel, we put them under a **PAR**. Remember that everything underneath a **PAR** gets indented by two spaces!

```
PAR
  button.press (2)
  pin.toggle (12, LOW)
```

We're only missing one thing at this point: the **CHAN**nel. We have to **declare** the channel outside the **PAR**. What this means is that we have to tell occam-π that we want a communications channel (we'll call it `s`). Further, we have to tell occam-π what *kind* of information it will carry. In this case, we want it to carry information of type **SIGNAL**.

The declaration looks like this:

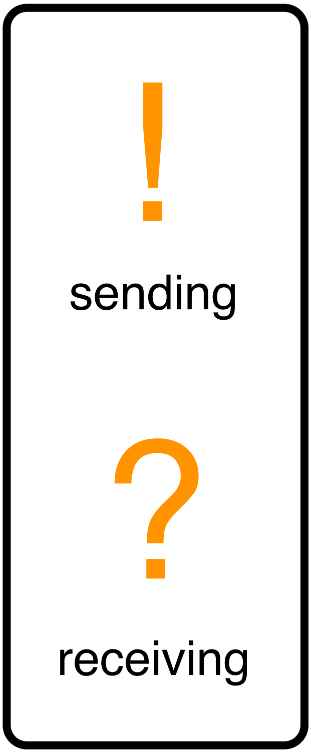
```
CHAN SIGNAL s:
PAR
  button.press (2)
  pin.toggle (12, LOW)
```

By placing the declaration before the **PAR**, it indicates that the processes that are running in parallel may, if they wish, use the channel `s` to send a **SIGNAL**. Now we have just one step left: we have to give one end of the channel to each of the two processes in the **PAR**.

A channel is a lot like soup cans on string: one person talks, and the other listens. Unlike soup cans on string, you can't take turns talking and listening at both ends: there's just one talking end, and just one listening end, and that's it.

The talking (or **sending**) end of a channel is marked with a `!`. The exclamation point (or *bang*, from British English) tells occam-π which end of the channel is sending information. It's as if that end of the channel is shouting: "Hey! Hey you! I've got a **`SIGNAL`** for you!"

The listening (or **receiving**) end of a channel is marked with a `?`. The question mark (or *eh?*, from Canadian English) tells occam-π which end of the channel is receiving information. It's as if that end of the of the channel is waiting for instructions: "Eh? Come again? What did you say?"



Lastly, wrap your code in a **PROC**edure called `main`, and indent everything another two spaces. (We don't get that step from the diagram.) When you're done, your program should look like this:

```
PROC main ()
  CHAN SIGNAL s:
  PAR
    button.press (2, s!)
    pin.toggle (13, LOW, s?)
:
```

Figure 5.1: Using a button to toggle an LED on and off.

5.3.2 In summary

To summarize what we learned in this chapter:

- Each box in the diagram represents a **PROC**edure. Because there are two boxes in the diagram, we can expect to find a **PAR** in our code with two processes indented underneath it.
- The two processes are connected by a single **CHAN**nel. A channel carries information in one direction only, from one process to another. In this chapter, the channel's name was `s`.
- Each process may take some parameters; in this chapter, the process called `button.press` takes a pin number; the process `pin.toggle` takes both a pin number as well as whether it should start out **LOW** or **HIGH**.

5.4 Breakage

There are lots of ways to break the code in this chapter, as usual.

Forget the colon
: At the end of the `CHAN`nel declaration, remove the colon. What happens? This is a common error made by many.

Indent the PAR
: What happens if you indent the `PAR` two spaces under the `CHAN`? We suspect this makes Plumbing unhappy. This is rather common as errors go as well.

Swap `!` and `?`
: Try switching the location of the `!` and `?`.

Leave out the `!` and `?`
: Try it. As it happens, this isn't an error. Perhaps it should be?

Forget the `SIGNAL`
: When we write `CHAN SIGNAL s`, we are telling Plumbing that the channel carries information of the type `SIGNAL` over the channel `s` and nothing else. What happens if you delete the word `SIGNAL`?

Forget the `s`
: When we write `CHAN SIGNAL s`, we are telling Plumbing that the channel is named `s`. What happens if you remove the letter `s`?

6 Tick... tick... tick...

The neat thing about a process network is that you can swap one process for another if it "speaks the same language." By this, we mean that you can swap one process for another if the input channels and the output channels carry the same kind (or type) of information.

In this short chapter, we'll make one change to the code from Push the Button to demonstrate this.

6.1 Seeing the parts

We just finished writing our first process network; now, lets tear it apart. This network was made up of two processes, `button.press` and `pin.toggle`.

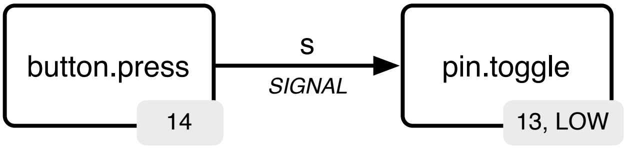
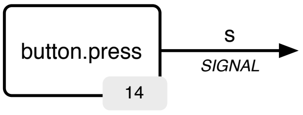
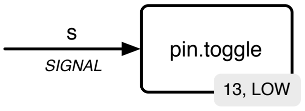


Figure 6.1: Our two-process network.



(a) One output channel...



(b) ... or one input channel.

Figure 6.2

If we look at this process network from the point of view of `button.press`, we might say:

> `button.press` has one **output** channel, `s`, that carries information of type **SIGNAL**.

We can also look at this network from the point of view of `pin.toggle`. Then, we might say:

> `button.press` has one **input** channel, `s`, that carries information of type **SIGNAL**.

We can see these two ways of looking at the parts of our process network in Figure 6.2. Now we'll introduce a new new **PROC**edure from the Plumbing library: `tick`.

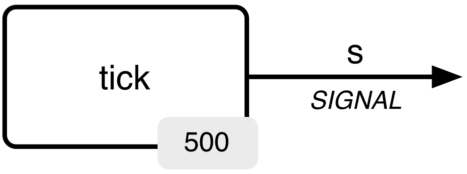


Figure 6.3: Another **PROC**edure in the Plumbing library.

The process `tick` takes two parameters: the first is the number of milliseconds between ticks, and the second is the sending end of a channel that carries signals. Now, here's the code from Chapter 5:

```
PROC main ()
  CHAN SIGNAL s:
  PAR
    button.press (2, s!)
    pin.toggle (13, LOW, s?)
:
```

Figure 6.1: Our original program uses `button.press`.

Because both `button.press` and `tick` have the same inputs (none) and the same outputs (one channel that carries messages of type **`SIGNAL`**), we can substitute one for the other. Once we do that substitution, our code looks like:

```
PROC main ()
  CHAN SIGNAL s:
  PAR
    tick (500, s!)
    pin.toggle (13, LOW, s?)
:
```

Figure 6.2: One substitution changes the program.

Because the `pin.toggle` process expects a **`SIGNAL`** to tell it when to turn its pin on or off, it doesn't matter where that signal comes from. It could be that it comes from a button press, or it could come from a clockwork ticker!

6.2 Exploring "plug-n-play"

As we continue to explore the Plumbing library, you are encouraged to experiment with modifications of your existing process network. Particularly, we hope you explore this notion of what we call "plug-n-play."

Process networks are connected by **CHAN**nels. As you saw in this chapter, we can get very different behavior from very similar code, simply by replacing one process with another. When using occam-π and the Plumbing library, your should strive to write lots of small, simple **PROC**edures. Then, you can combine these **PROC**edures into a process network, and (most importantly) rearrange the process network (or substitute one process for another) if you want to get different behavior from your program.

For example, if you are developing a piece of art that you intend to be interactive, you might start by testing it with a process like `tick`. Then, when you're done programming, you might switch that with a process that handles input from the world (eg. `button.press`). Then, your piece is ready for interaction with people viewing your piece in a gallery.

Matthew's Runaway was a piece of art developed by substituting one **PROC**edure for another.

7 Undressing Toggle

In the last chapter, we saw that one process can be substituted for another when they have the same input and output channels. In this chapter, we'll explore how we can combine several **PROC**edures into one new **PROC**edure, simplifying our code further.

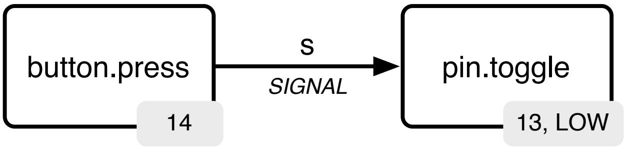
Because we promised that you would learn how to write everything you saw in this book, we will peel back the layers of the `pin.toggle` procedure, and show you what is inside of it. Specifically, you'll see that the `pin.toggle` procedure is just two *more* procedures stuck together with a channel!

7.1 The Circuit

The circuit for this chapter is identical to Chapter 5. You should be able to type in the code from this chapter, run it, and get exactly the same behavior as before.

7.2 The Network

We'll continue using the network from Chapter 5: Push the Button.



7.3 Breaking up is hard to do

We're going to "break up" the process called `pin.toggle`, and represent it as two processes instead of one. The result will be a process network with a total of three processes.

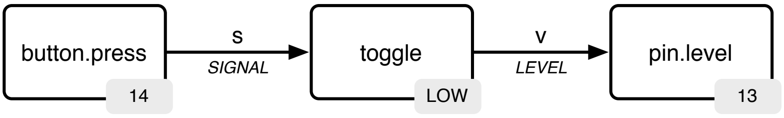


Figure 7.1: The process network for this chapter's code.

If you were to look in the code for the Plumbing library, you would find that `pin.toggle` is actually just two processes running in **PAR**allel with each-other. In this chapter, we'll break `pin.toggle` apart, and in the next chapter, we'll put it back together again.

7.3.1 From pictures to code

From this network, we can write the code. We know there are three processes, so we'll list all of them. Further, we know they communicate with each-other (because they're connected with channels), so we'll put them under a **PAR** right from the start.

```
PAR
  button.press ()
  toggle ()
  pin.level ()
```

Next, we can see there are two **CHAN**nels. One is of type **SIGNAL**, the other of type **LEVEL**. To use a channel, we have to declare it before it is used. So, we'll include *two* channel declarations.

```
CHAN SIGNAL s:
CHAN LEVEL  v:
PAR
  button.press ()
  toggle ()
  pin.level ()
```

Next, we need to give the **PROC**edures their parameters. We do this through a combination of reading the Plumbing library documentation[1] as well as studying the process network diagram.

[1] `http://concurrency.cc/docs/`

`button.press` takes a pin number and the sending end of channel, `toggle` takes an initial level (which we set to **`LOW`**), two channel ends (the receiving end of `s` and the sending end of `v`), and `pin.level` takes a pin and the receiving end of `v`.

```
CHAN SIGNAL s:
CHAN LEVEL  v:
PAR
  button.press (2, s!)
  toggle (LOW, s?, v!)
  pin.level (12, LOW, v?)
```

The last step is to wrap everything up in a `main()` **`PROC`**edure. This is the same thing we did in Chapter 5.

```
PROC main ()
  CHAN SIGNAL s:
  CHAN LEVEL  v:
  PAR
    button.press (2, s!)
    toggle (LOW, s?, v!)
    pin.level (12, v?)
:
```

Once again we have studied a process network and converted it into code based on our understanding of what the diagram means.

7.4 What does `toggle` do?

Toggle is new to us in this chapter. It takes in a **`SIGNAL`**, and outputs a **`LEVEL`**. If you focus your attention on `toggle` only (Figure 7.2), you'll see that it has one channel coming in from the left, and one channel going out on the right. The `toggle` process waits until it receives a **`SIGNAL`** on the channel `s`; when it does, it sends out a message on the channel `v`.

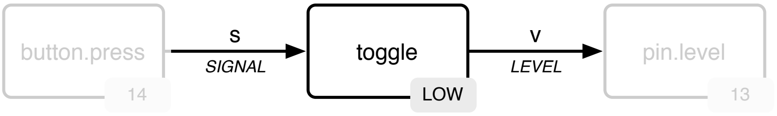


Figure 7.2: Toggle has one channel in, and one channel out.

`toggle` is kinda cool. Up until this chapter, we have only seen **`SIGNAL`**s. Now, we have **`LEVEL`**s. Digital pins on your Arduino can be set to either **`HIGH`** or **`LOW`**. When they are **`LOW`**, they are off. When they are **`HIGH`**, they are at +5 volts. `toggle` starts off **`LOW`**, and every time it receives a **`SIGNAL`**, it flips its level and sends that value out.

We could say that `toggle` follows the following three steps over and over:

1. `toggle` waits until it receives a **`SIGNAL`**,
2. flips the value of its level,
3. sends out its new level, and
4. goes back to step 1.

If we "unroll" this loop a bit, we would say that `toggle`:

1. waits for a **`SIGNAL`** on channel `s`,
2. flips from **`LOW`** to **`HIGH`**,
3. sends **`HIGH`** out on channel `v`,
4. waits for a **`SIGNAL`** on channel `s`,
5. flips from **`HIGH`** to **`LOW`**,
6. sends **`LOW`** out on channel `v`,
7. ...

You could say that `toggle` turns **`SIGNAL`**s into messages that have a value, and those values are used to turn a pin on (**`HIGH`**) and off (**`LOW`**).

7.5 Pattern: A Pipeline

Plumbing programs are all about networks of processes sending information to each-other using channels. Channels are (we're sorry) the *plumbing* that makes our programs work. In the previous chapter, you saw how `button.press` sent a **SIGNAL** to the process `pin.toggle`, which then turned the built-in LED on and off. As it turns out, this whole network of processes is like a *pipeline* that carries information from one stopping point to the next.

A pipeline carries stuff from one place to another. When we string multiple processes together in a straight line, we have constructed a **pipeline** of processes. It is, essentially, an assembly line, where each process processes some information, and then passes the result of that work along. In Plumbing, each process has to wait for information from the "upstream" process; that is, `pin.level` waits for messages from `toggle`, and `toggle` waits for messages from `button.press`. In technical terms, this would be called a *buffered, synchronous* pipeline.[2].

The Pipeline is one of the simplest patterns for processing information in parallel. It shows up everywhere, from the command-line on Linux, to the way a web server handles requests from your web browser, to the central processing unit in your computer. The Pipeline is one of the Big Ideas in computing.

Congratulations. You've just explored it on your Arduino.

[2]See `http://en.wikipedia.org/wiki/Pipeline_(computing)` for more information about pipelines in computing.

7.6 Explorations and Breakage

One thing you can do is to explore the Plumbing library a bit at this point. There are many processes in the Plumbing library; one you haven't seen yet is called `invert.level`. It takes in a **`LEVEL`** value on one channel and outputs the opposite value on another. That is, it has a **`LEVEL`** channel coming in, and a **`LEVEL`** channel going out. Try using it in your process network—it only fits in one place.

Or, you can explore how you can break this code. It may seem like we come up with lots of ways to break your code at the end of every chapter. That's because we want you to experience as many errors as possible in a controlled way before we set you free.

Wrong channel types
: What happens if you take the code we started with and swap around **`SIGNAL`** and **`LEVEL`** on lines 2 and 3? Do things still work?

Wrong channel order
: Modify line 7 so that `toggle` has its read and write channels in the wrong order. That is, flip it from `(s?, v!)` to `(v!, s?)`.

Wrong channel directions
: Modify line 7 so that `toggle` has its read and write channels are pointing in the wrong direction. That is, flip it from `(s?, v!)` to `(s!, v?)`. (Subtle, no? Don't worry. If you draw pictures first, this is a difficult mistake to make.)

Wrong pin number

What happens if you have the wrong pin number in either `button.press` or `digital.output`?

Initial state flip

What happens if you change `digital.output` so that it starts with **`HIGH`** instead of **`LOW`**?

Forgetting a process

What happens if you simply remove `toggle`? (Draw a new version of the process network from this chapter, and leave out `toggle`. Does that look right? This is what happens when you remove line 7 from your program!)

8 Buttons Everywhere

When you're developing your program—exploring ideas—you'll often find that things get long and out-of-hand. This chapter will help you see both how to handle multiple inputs and multiple outputs, as well as how to reorganize your code to make it more readable.

8.1 The Challenge

You want to develop a circuit that has multiple inputs and multiple outputs. In our case, we'll use pushbuttons as our inputs and and LEDs as our outputs—but the principles we're going to explore will be the same regardless of what serves as the source for your input and the destination for your output.

We know that one button and LED yields a process network with two **PROC**edures. This will serve as our starting point.

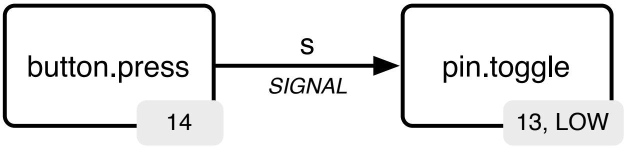


Figure 8.1: The network from Chapter 5 again.

8.2 The Circuit

This circuit builds on the circuit from Chapter 5. We add another button (to pin 3) and another LED (to pin 6).

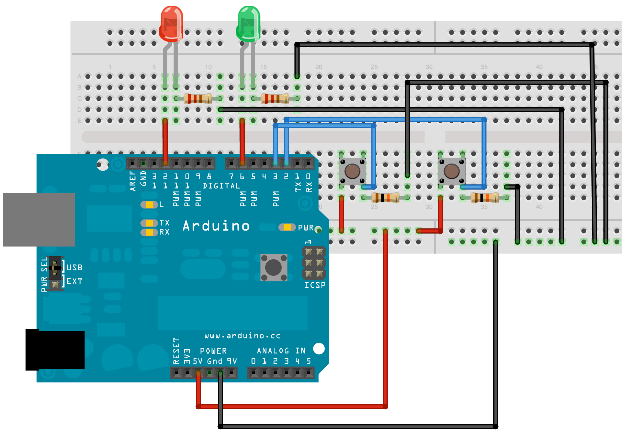


Figure 8.2: Two buttons, two LEDs.

Looking at the process network on 8.1 on the preceding page, we can see that it will work just fine with this circuit. That is, we can run the code from the previous chapters and still control one of the LEDs. It we want to handle the other button and LED, however, we'll need another `button.press` procedures and more `pin.toggle` procedures.

8.3 Reusing Procedures

Here is the code from Chapter 5.

```
PROC main ()
  CHAN SIGNAL s:
  PAR
    button.press(2, s!)
    pin.toggle(13, LOW, s?)
:
```

We have one channel `s` connecting our button input from pin **2** to our toggle process controlling the LED on pin **13**. As it happens, we're able to use **PROC**edures more than once, providing them with different parameters. In a picture, the process network looks like Figure 8.3 on the next page.

Put simply, we run the same **PROC**edure multiple times, but we provide it will different parameters. One `button.press` is told to watch pin **2**, while the second `button.press` is told to watch **3**. Note, though, that each **CHAN**nel must have its own unique name. Although it isn't very creative, I've named one channel `s1` and the other `s2`.

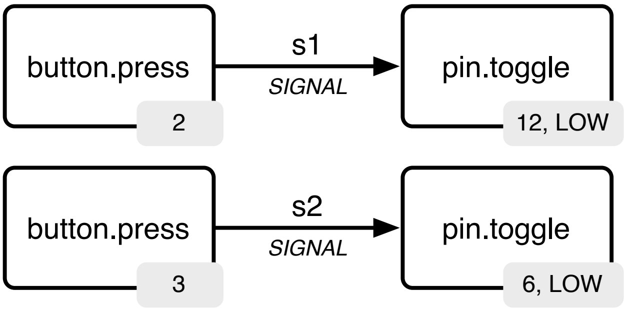


Figure 8.3: Change the parameters to reuse **PROC**edures

The reason we like using occam-π for writing this kind of program is that it is remarkably easy to convert this process network into code. We take the code we wrote before, and we simply run two more processes under the same **PAR**. We will have to add one more channel declaration, but that's not so hard. When we're done adding in two more processes, we end up with code that looks like this:

```
PROC main ()
  CHAN SIGNAL s1, s2:
  PAR
    button.press(2, s1!)
    pin.toggle(13, LOW, s1?)
    button.press(3, s2!)
    pin.toggle(6, LOW, s2?)

:
```

8.4 Managing complexity

When programming in occam-π, we like the fact that we can add more **PROC**edures underneath a **PAR** and handle more things concurrently ("at the same time"). Unfortunately, our **PAR** can grow a bit unwieldy. Eventually, it's nice to be able to bundle things up and shrink the amount of code we have in any one **PROC**. If you prefer, we're going to reverse the process we saw in the last chapter: instead of breaking one process apart, we're going to build a new one by putting smaller pieces toether.

For example, it might be nice to introduce a new **PROC** called `b2p` (short for "button to pin"). This **PROC**edure will take two parameters: the pin that a button is connected to (`button.pin`), and the pin that an LED is connected to (`led.pin`). It will "hide" from us the fact that there is a channel communication between `button.press` and `pin.toggle`.

We might start our code this way:

```
PROC b2p (VAL INT button.pin, led.pin)
:
```

The two parameters that this **PROC**edure is going to expect are **constants**, or numbers that cannot be changed. In occam-π, we can't say `CONSTANT`, but we can say `VAL INT`, which means it is a number (an **INT**eger) and it is a **VAL**ue, not a variable.

Inside of this **PROC**, we can build a small process network. Specifically, we'll put `button.press` and `pin.toggle`, and connect them up as we did before.

```
PROC b2p (VAL INT button.pin, led.pin)
  CHAN SIGNAL s:
  PAR
    button.press(button.pin, s!)
    pin.toggle(led.pin, LOW, s?)
:
```

The critical change made in connecting `button.press` to `pin.toggle` is that we substituted `button.pin` for one of the parameters of `button.press`, and `led.pin` for one of the parameters of `pin.toggle`.

Now that we have combined `button.press` and `pin.toggle` in one **PROC** called `b2p`, we can use that in the rest of our code. We can now write our `main` **PROC** this way:

```
PROC main ()
  PAR
    b2p(2, 12)
    b2p(3, 6)
:
```

If we were to "peel back" both of these **PROC**edures, we'd end up right back where we started: with two copies of the `button.press` **PROC** and two copies of `pin.toggle`, each connected by their own channel. This way, we've simplified our program, and we can now reuse `b2p`, our new **PROC**, as part of other process networks.

8.5 The Code

Here is the full program from this chapter, in one place.

```
PROC b2p (VAL INT button.pin, led.pin)
  CHAN SIGNAL s:
  PAR
    button.press(button.pin, s!)
    pin.toggle(led.pin, LOW, s?)
:

PROC main ()
  PAR
    b2p(2, 12)
    b2p(3, 6)
:
```

8.6 Breakage

As always, there are many ways to break your code. Breaking code is part of how you learn.

Bad constants

What happens if you use a value for a pin that is too large (eg. `42`)?

Forget the `VAL`

What happens if, in your definition for `b2p`, you forget the word `VAL`?

Reuse some pins

What happens if you try and use `b2p` twice, but you use the same constants both times?

Mixup pins
: What happens if you mix up `button.pin` and `led.pin` in the **PROC**`b2p`?

Switch the order
: Does anything break if you put one `b2p` before the other in the **PAR**?

Forget a comma
: There are several place that we've used commas. What happens if you leave one out?

Use a channel twice
: If you go back to the code from the middle of the chapter, what happens if you use `s1?` more than once? What happens if you use `s2!` more than once?

Fail to use a channel
: What happens if you declare a channel but never use it? Add

```
CHAN SIGNAL oops:
```

before the **PAR** in your `main` and see what happens.

9 Making things Move: Servos

So you've got a servo, do you? And you want to drive it with your Arduino, do you? Well. You've come to the right place. In this next series of chapters we'll be walking through the code and syntax you'll need to drive servos. In this chapter we're going to simply explore turning the servo on, but we'll quickly be moving up towards more interesting operations. It's also definitely worth a note that even if you don't have a servo, you can still explore the concepts and syntax brought forward in these next five chapters. The simple.servo(..) process we'll be spending a great deal of time with has the same signature as the pwm(..) process that was the focus of chapters [whatever], so if you're willing to experiment some, all of the code that's designed here to drive servos can drive a fading LED!

9.1 Goals

1. To understand what a servo needs in order to properly operate.

2. To revise our understanding of what CHANnels can do.
3. To set a servo's position with the simple.servo(..) process.

9.2 Building the Circuit

One of the fun things about this circuit is that we don't actually need the bread-board. We can plug our servo right into the Arduino pins and run straight off the board. All you will need are: 1 Your Arduino 2 Your Servo [[CIRCUT DIAGRAM]]

You servo might not look much like the one in the diagram above, and perhaps your ground, power and control wires aren't in the same order. That's perfectly fine. What's crucial is making sure your ground wire is the one plugged into the ground pin, your power is plugged into the 5v pin, and that your control is plugged into one of the correct PWM pins. If this is not done, bad things could very well happen to both your board and your servo. And that would be bad. Measure twice, run your servo.. more than once, actually, is the goal.

The fact that we're plugging the control wire into pin 11 is more important than you might think. As discussed in previous a chapter[s], PWM pins are connected to Counters in the Arduino's processor. Some of these counters are 8 bit (and can thus only count up to 255 – the maximum value of a single BYTE), and – on the Arduino – only one of these is a 16 bit counter (meaning it can count well past 60,000). For servos to operate properly, we need the control wire to be plugged into not just a PWM pin, but a 16 bit PWM pin.

[[SIDE BAR]] On the Arduino, and any other board using an ATmega328p processor, the only two pins that are connected to a 16 bit timer are: 9, and 10. On boards like the ArduinoMega and the ArduPilotMega – boards equipped with the Atmega1280 – the range of 16 bit pins is quite a bit wider. You can select from pins: 2, 3, 5, 6, 7, 8, 11, 12, 13, 44, 45 or 46.

9.3 Code

```
PROC main ()
  CHAN BYTE pos.chan:
  PAR
    simple.servo (11, pos.chan?)
    pos.chan ! 45
:
```

Before we get started talking about the code we'll be writing today, there's one piece of information you probably want to have on hand. We have an appendix in the back of this book that describes in enough detail the mechanical operations of servos, but as teaching electrical engineering isn't the goal of this book, we won't bore you with unnecessary details. What you need to know though, is that the /minmum and maximum control pulse widths are to 1000 mircoseconds and 2000 mircoseconds respectively/. As each servo is different, yours might have different minimum and maximum values. If your servo can take shorter and longer pulse widths (as is probably the case), there's nothing to worry about. If your servo's minimum pulse needs to be longer than 1000 mircoseconds, or if its maximum pulse needs to be shorter than 2000, *this code may be damaging to your servo*. It's always better to be safe than sorry, so if you love that servo you're going to be playing with, be sure to double check its speicfications. If need be, simply read through these next five chapters so you have an idea of what's going on with the code, and when you get to chapter six we'll shoe you how to set the minimum and maximum pulse widths manually.

As has already been mentioned, the hot, new processes we're going to be spending some time with in this section is simple.servo(..). It's a very simple process. As we can see from the network digram for the code we're about to write, simple.servo(..) only has one parameter, and takes a single CHANnel of type BYTE

[[PARTIAL NETWORK DIAGRAM]]

CHANnels of BYTEs are built to carry a good bit more information than the channels we're used to working with. SIGNAL channels actually only convey the information that a SIGNAL has been sent. There's no state data that's sent in a SIGNAL. We have played a little bit with LEVEL channels which do carry some information with them, but LEVEL is a very limited data type. It can contain either HIGH or LOW, and nothing beyond that. The BYTE data type can contain any number between 0 and 255 which – as far as numbers are concerned – isn't that much of a range. It's enough to contain the degree ranges of 0 to 180 though, which should act as a good approximation for the range of most hobbie servos. You can see the BYTE channel we're going to be using is named pos.chan, which is – if you fcan believe it – short for position channel, as this channel is what we're going to be using to set the position of our servo.

So, to get some code written from that network diagram, we can see that we're going to have to declare a CHANnel, and place simple.servo in parallel (so communications can occur). [[CODE]] CHAN BYTE pos.chan: PAR simple.servo(11, pos.chan?)

If you haven't started worrying about what's missing been missing from that network diagram... well, don't worry, we're about to solve that little conundrum. We've left off what's connected to the sending end of pos.chan because there actually isn't going to be a named process talking down that channel. So far in this book, we've always been plugging together pre-built process with channels; passing the sending end to one process and the recieving end to another. Because we want to actually retain control over the servo's positions, it won't do to hand the sending end of that channel to another process. We need to communicate down that channel ourselves.

The way to do that is remarkably easy, actually. There are three elements to any message sent directly by the user. The first thing we have to do is specifiy which channel we're going to talk down, [[CODE]] *pos.chan* Tell our code that we're going to be sending a message down the channel (if you remember, sending and recieving has everything to do with the ! (bang) and the ? ('eh). We've already passed the ? into simple.servo, so we're going to be taking advantage of that ! now) [[CODE]] pos.chan *!* And then we need to specify what we want to send. Becuase this channel is controlling the position of our servo in terms of the degrees between 0 and 180, let's tell that servo to head towards the 45 degree mark. [[CODE]] pos.chan ! *45*

[[SIDE BAR?]] It's very important that we always make sure that the message we send is of the same type as the channel we're sending it down. Channels of type LEVEL can only send HIGH or LOW, as those are the only two LEVELs out there. Channels of type BYTE, as has been said, can only send the numbers between 0 and 255, etc. SIGNAL channels are a little special though. I said that every communication down a channel is made up of those three elements – the channel name, the fact that we're sending something, and what we're communicating – but because SIGNALs don't actually communicate any state data – just that a SIGNAL has been sent – there's nothing for a SIGNAL to communicate. To send a SIGNAL, we just have to write: [[CODE]] signal.name ! [[/SIDE BAR?]]

Once we've updated our network diagram to show the fact that we're going to be the ones communicating down pos.chan:

[[UPDATED NETWORK DIAGRAM]]

We can see that we need to add that line of code that sends that positinal information under the PAR so simple.servo can hear us when we ! that message off. [[CODE]] CHAN BYTE pos.chan: PAR simple.servo(11, pos.chan?) pos.chan ! 45

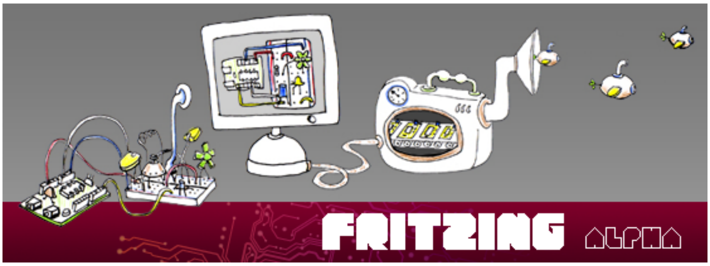
Then we just need to wrap that code in the usual PROC main () ... :, indent everything those last two spaces, and we'll have some code that sets our servo to a 45 degree angle! [[CODE]] PROC main () CHAN BYTE pos.chan: PAR simple.servo(11, pos.chan?) pos.chan ! 45 :

10 Acknowledgements

10.1 Software

Plumbing for the Arduino was typeset using LaTeX. Writing was carried out in either TextMate by Allan Odgaard or vi. Diagrams were produced using OmniGraffle Pro by the Omni Group, and screen captures were made using Snapz Pro by Ambrosia Software.

The *completely awesome* circuit diagrams were made using Fritzing, an open source project that lets complete noobs design circuits visually, then see the same circuit as a schematic, and finally export that circuit as a PCB for etching by hand or automated manufacture.



Fritzing is truly a wonderful tool in progress; it was one of our discoveries while working on Plumbing. We had to to take a moment and gush its praises here—explore it, and discover the joy of circuit design.

10.1.1 occam-π and Plumbing

occam-π was originally designed under the guidance of David May, implemented by the fine people at inmos, and shepherded by Professor Peter Welch at the University of Kent in Canterbury, England for the past 20 years (give or take). Many of the features that put the π in occam-π were implemented by Fred Barnes.

The Transterpreter, which allows us to run occam-π on tiny platforms like the Arduino, was originally designed and written by Christian Jacobsen and Matt Jadud, and extended and improved by Damian Dimmich, Carl Ritson, Adam Sampson, and Jon Simpson. The `concurrency.cc` board was designed by Omer Kilic. The `concurrency.cc` logo was designed by Geoffrey Long.

The Plumbing library was originally written by Christian Jacobsen, Matt Jadud, and Adam Sampson. Contributors since then include:

Radu Creanga Code for implementing PWM.

10.2 Images

All images in this text were produced by the authors unless noted below. We have tried to use images placed in the Commons wherever possible.

The cover image was made available under a CC-BY license by Flickr user *macinate*:

`http://www.flickr.com/photos/macinate/2191054677/`

The T414 on page 8 was found on the Wikipedia under a CC-BY-SA-2.5 license:

`http://en.wikipedia.org/wiki/File:IMST414B-G20S.JPG`

The juggling LEGO figure on page 46 was made available by Flickr user *helico* under a CC-BY license:

`http://www.flickr.com/photos/helico/404640681/`

The image for "waiting" on page 50 was made available by Flickr user *red twolips* under a CC-BY license:

`http://www.flickr.com/photos/25182350@N03/2957915812/`

The image for "pipeline" on page 70 was made available by Flickr user *nz_willowherb* under a CC-BY license:

`http://www.flickr.com/photos/willowherb/3320805930/`

The image for delta PWM on page **??** was made available by Cyril Buttay under a CC-BY-SA license:

`http://en.wikipedia.org/wiki/File:Delta_PWM.png`

The FTDI chip image on page 12 was made available as part of the Arduino "Getting Started" guide under a CC-BY-SA 3.0 license (20110115):

`http://arduino.cc/en/uploads/Guide/FTDIChip.png`

The Arduino Pro without an FTDI chip on page 13 was made available by David Mellis under a CC-BY 2.0 license (20110115):

`http://www.flickr.com/photos/mellis/4784333335/`

The FTDI Friend on page 14 was made available by Lady Ada under a CC-BY license (20110115):

`http://www.flickr.com/photos/ladyada/4987941401/`

11 Book Bugs

Our text is not perfect. If you find errors in the text, please pass them on to `bookbugs at concurrency dot cc`. We would like to acknowledge your help here.