Bitlash User's Guide

For Bitlash 2.0

Bitlash User's Guide
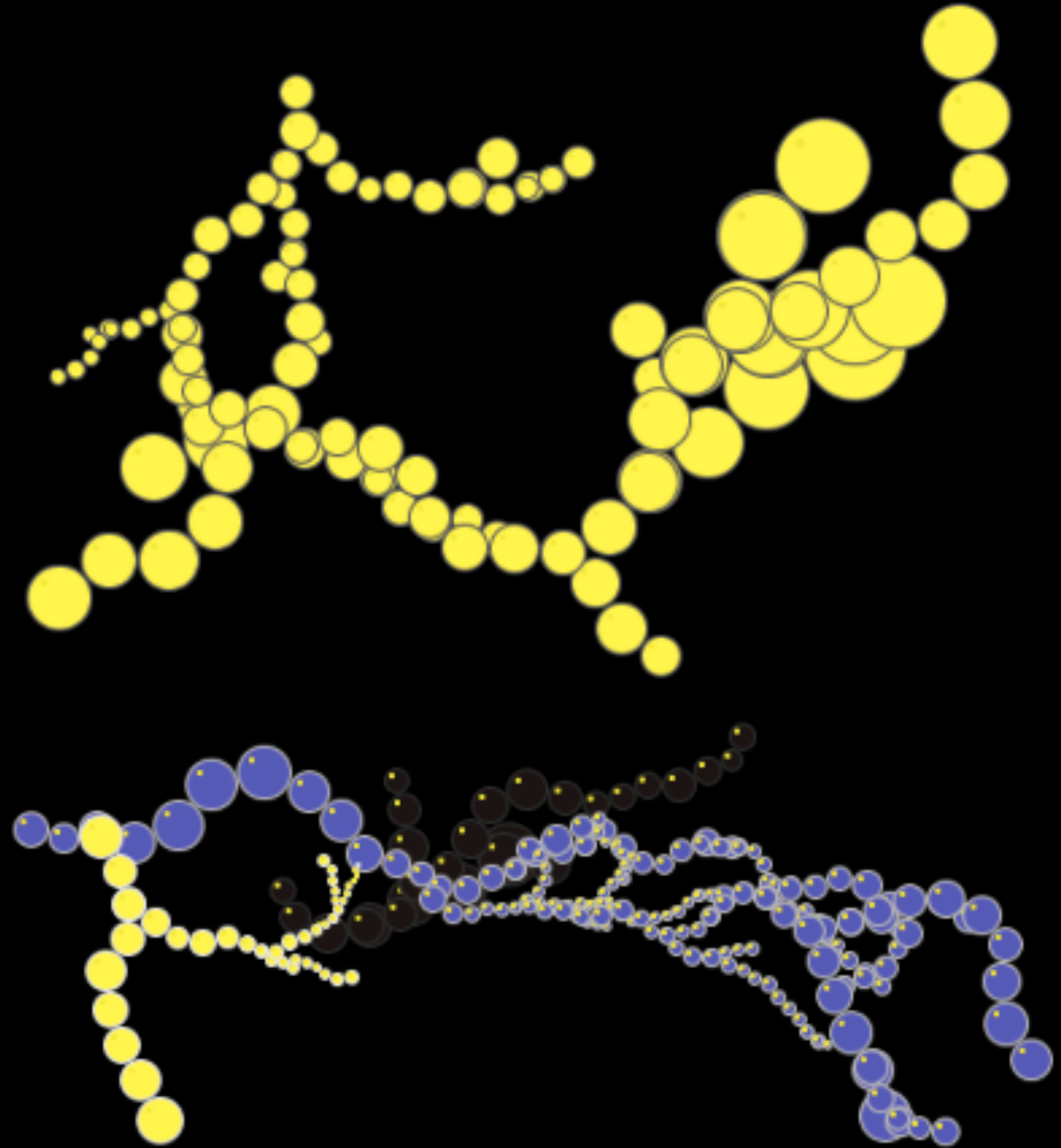
About Bitlash

What is Bitlash?

How does it work?

Bitlash architecture

What is Bitlash?

BITLASH IN A NUTSHELL

1. **Interprets a C-like mini-language extended for digital and analog IO**

2. **Recognizes most Arduino library functions**

3. **Runs entirely on the Arduino with serial IO for control and output**

4. **Saves functions defined from the command line in EEPROM; small applications are possible**

5. **Mini-tasker runs functions periodically in the background for system control**

6. **API makes it extending Bitlash with your C code easy**

7. **Open Source on Github (http://bitlash.net)**

8. **Extensive documentation and sample applications**

Bitlash is an open source interpreted language shell and embedded programming environment for the popular and useful Arduino.

The Bitlash interpreter runs entirely on the Arduino and interprets commands that you type in a terminal window or send programmatically to the serial port:

```
bitlash here! v2.0 (c) 2012 Bill Roy -type HELP- 942 bytes free
> print "Hello, world!", millis()
Hello, world! 11939
>
```

It's easy to extend Bitlash by naming a sequence of commands as a Bitlash function, which is like a new word in the Bitlash language. A Bitlash application is a set of such functions, and since they are stored in the EEPROM, Bitlash can start up your application automatically at boot time for standalone operation.

In cases where you need native C code to get closer to the hardware, perhaps for speed or to interface with a custom peripheral, it's also easy to integrate your C code with Bitlash as a User Function. Bitlash ships with many examples of user functions.

How does Bitlash work?

BITLASH INTERPRETER

1. **Runs in under 16kB on the AVR '328**

2. **Parses Bitlash code on the fly from RAM, PROGMEM, EEPROM, or SD card**

3. **Executes commands live from the command line and runs background tasks**

The Bitlash embedded interpreter that runs in about 14k of memory on an Atmel AVR processor. It works nicely with Arduino to make a development and prototyping tool for those situations where you need to bang some bits on the Arduino but writing a sketch in C is premature. It's very convenient for bringing up and debugging new hardware.

The Bitlash command language is very similar to Arduino C and includes a large repertoire of the familiar Arduino C functions so you can hack your hardware from the serial command line or even over the internet via telnet.

You can store commands as functions in EEPROM, compile them into flash memory, or put them on an SD card. A user-defined bitlash startup function is run automatically at bootup, making the automation and maintenance of small applications very easy. Here is the classic Blink13 program as a complete Bitlash application in two functions, **toggle13** to toggle the led and **startup** to initialize and run it:

```
> function toggle13 {d13=!d13;}
> function startup {pinmode(13,1); run toggle13,1000;}
> boot
(d13 toggles every 1000ms)
```

Bitlash architecture

ARCHITECTURE BASICS

1. **Bitlash is an Arduino sketch, written in C**

2. **Bitlash uses the Arduino and avr-gcc libraries**

3. **Bitlash can use user C code and libraries**

4. **These libraries in turn control the Atmel hardware**

Here is a diagram of the components of Bitlash and how they interoperate with the Arduino software and the Arduino hardware with Atmel CPU.
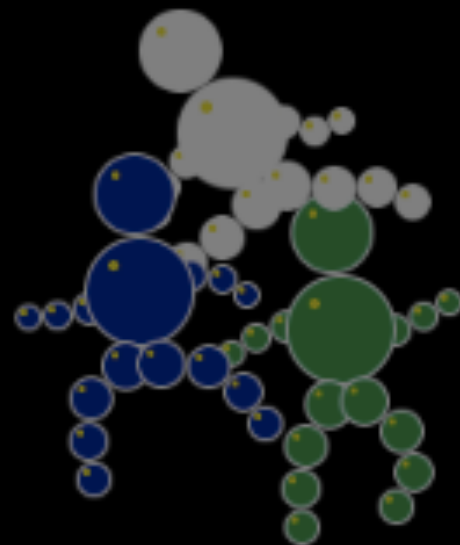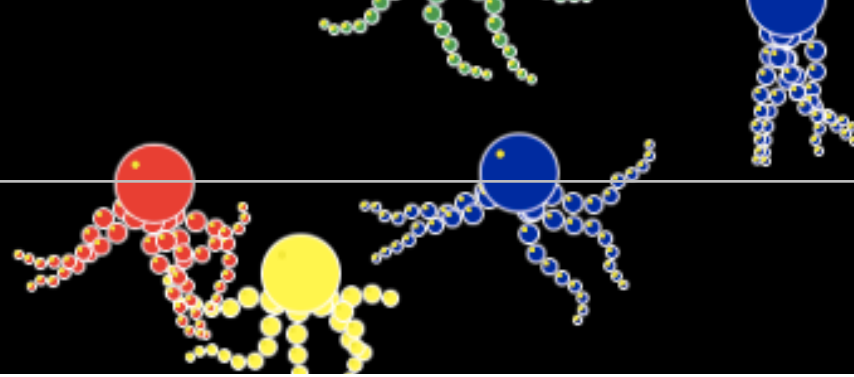

Bitlash on Arduino: Architecture Overview

Version: 1.0
Copyright 2010 by Bill Roy
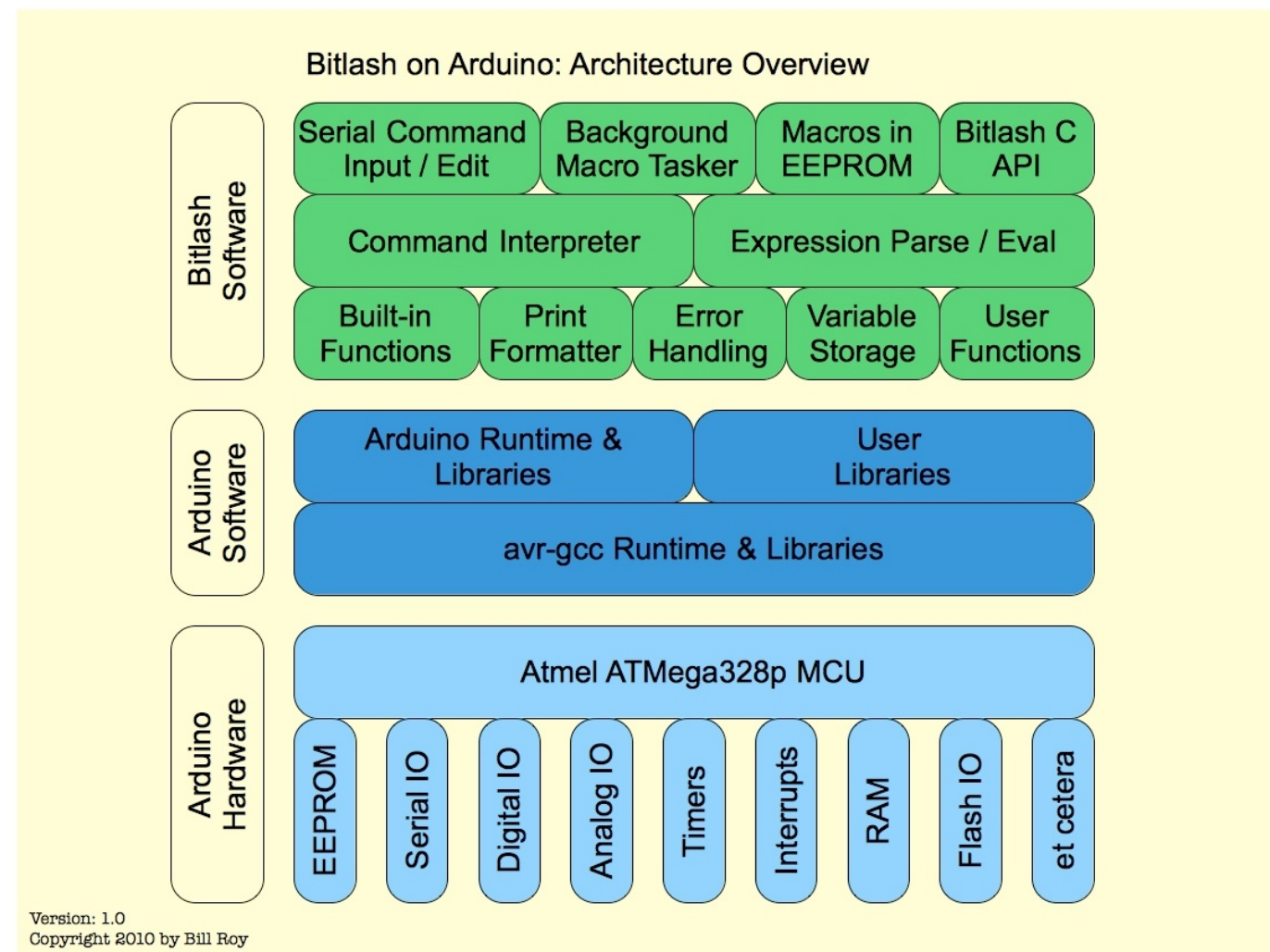
Get Bitlash

Download

Install

Hello, world!

Get Bitlash

1. Visit **http://bitlash.net** to download or *git clone* the latest version

2. Install the bitlash/ folder into the libraries subfolder of your Arduino sketchbook

3. Restart the Arduino IDE so it notices the new library

4. Connect with a terminal emulator like the Arduino Serial Monitor

5. First program: *print "hello, world"*

6. *function hello {print "hello, world"}*

1. Download Bitlash

If you use git, it's easiest to clone Bitlash via git; see below.

For a .zip file download, the latest version of Bitlash can be found at http://bitlash.net

2. Install Bitlash

Requirements

You need an Arduino connected to a PC with a working Arduino IDE. These directions are for Arduino v0023 through 1.0.1.

Since Bitlash is an Arduino library you upload with a sketch, you need to be comfortable uploading sketches. Get this working first to save debugging headaches. There is plenty of help over at the Arduino Forums at http://arduino.cc/forum/

Install Bitlash using Git

You can also grab the latest development version using Git, and clone it right into your libraries directory using the recipe below.

NOTE: Back up your existing Bitlash directory first!

```
$ cd ~/Documents/Arduino/Libraries
```

```
$ mv bitlash/ bitlash-save/
$ git clone https://github.com/billroy/bitlash.git
```

Restart the Arduino IDE and you're good to go.

Install Bitlash

The Arduino Development Environment Guide specifies this procedure
for installing third party libraries:

```
"To install these third-party libraries, create a direc-
tory called libraries within your sketchbook directory.
Then unzip the library there."
```

There is also a post on the Arduino weblog with further explanation.

So, to install Bitlash,

- Create a directory called **librar-
  ies** within your sketchbook direc-
  tory
- Unzip the download and copy or
  move the Bitlash distribution
  folder into the **libraries** folder
  you just created
- Rename the resulting folder to
  simply **bitlash**, if necessary, to remove the version number. You
  should end up with a folder setup that looks like this:
- Restart the Arduino IDE; if all goes well, you will find the **bitlash**
  library listed in the **Sketch / Import Library** menu.
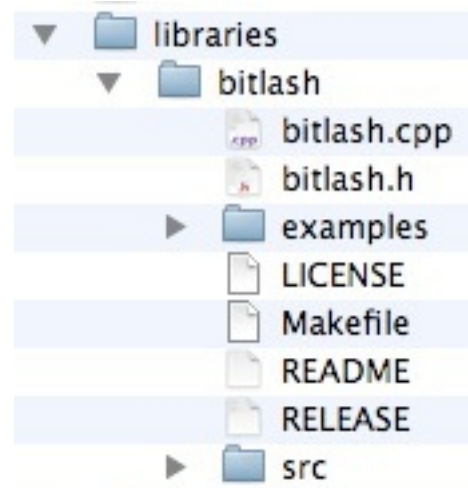
- Select **File / Examples / bitlash / bitlashdemo** to open the
  demo sketch, then **File / Upload** to compile and upload it to your
  Arduino.

When your upload is complete, you are ready to connect to Bitlash. Pro-
ceed to the next section on Connecting.

Connect With a Terminal Emulator

Connect to the serial port at 57600 baud using whatever terminal emu-
lator works for you. Here are some options:

- You can use the built-in Arduino Serial Monitor, but see the note
  below
- On Windows, HyperTerminal seems popular
- On OS X I use **screen**
- On Linux **screen** is available on most distributions

NOTE: USING BITLASH WITH THE ARDUINO SERIAL MONITOR

To use the Arduino "Serial Monitor" function with Bitlash, you must se-
lect "Carriage return" line ending handling option. The Bitlash demos
use a baud rate of 57600.

EXAMPLE: STARTING THE SCREEN COMMAND WITH BITLASH

Here is an example using the **screen** program in OS X to connect with
Bitlash on a USB-connected Arduino. The /dev/tty.usb… part is the vir-
tual serial port name that you can find in the Arduino Tools/Serial Port
menu and the 57600 is the baud rate:

```
$ screen /dev/tty.usbserial-A7003pQ3 57600
bitlash 2.0 here! (c) 2012 Bill Roy -type HELP- 935
bytes free
>
```

Congratulations, you are up and running: Bitlash is listening for commands, as signified by the '>' prompt.

Hello, World!

Now that you have a command prompt you can type a command, and press Enter.

Here is the usual Hello World example you might run as your first Bitlash program:

```
> print "Hello, world!"
Hello, world!
>
```

While you're there you might check the arithmetic:

```
> print 2+2
4
```

First App: Blink13

No discussion of "Hello, world!" for embedded systems would be complete without blinking an LED. This example shows how to build a complete Bitlash application using Bitlash functions and auto-start.

First it is necessary to introduce the concept of **pin variables**: Bitlash gives direct access to the digital IO pins via single-bit variables named

d0, d1, d2, and so on. You can read a pin variable's value and print it like this:

```
> print d12
0
```

…and assign it like this:

```
> d13=1    * turn on pin 13
```

…though you must remember to set the pin mode if you want the port to be an output:

```
> pinmode(13,1)
```

So, returning to blink13, what we want is to toggle the pin periodically. Let's define a function named **toggle13** to toggle the pin:

```
> function toggle13 {d13 = !d13;}
```

A function named **toggle13** containing the Bitlash code "d13=!d13;" is defined and saved in EEPROM. When the function **toggle13** runs, this program text sets pin d13 to the logical complement of its current value: if it was zero, it becomes one, and vice versa.

Now all we need is to arrange for **toggle13** to be run at the desired toggle rate, let's say every 1000 milliseconds; and let's not forget to set pin 13 as an output. By using the special function name **startup** we designate this function to be automagically run at boot time, completing our application:

```
function startup {pinmode(13,1); run toggle13,1000;}
```

List our functions to make sure they're right using the ls command:

```
> ls
function toggle13 {d13 = !d13;};
function startup {pinmode(13,1); run toggle13,1000;}
>
```

You can invoke the startup function from the command line to test:

```
> startup
(the LED on pin d13 is blinking)
```

You can also restart to test the power-on startup:

```
> boot
bitlash here! v2.0...
(the LED on pin d13 is blinking)
```

Next Steps: Learn Bitlash

Congratulations! If you get this far, you have a free-standing development environment on your Arduino.

Chapter 3

The Bitlash Language
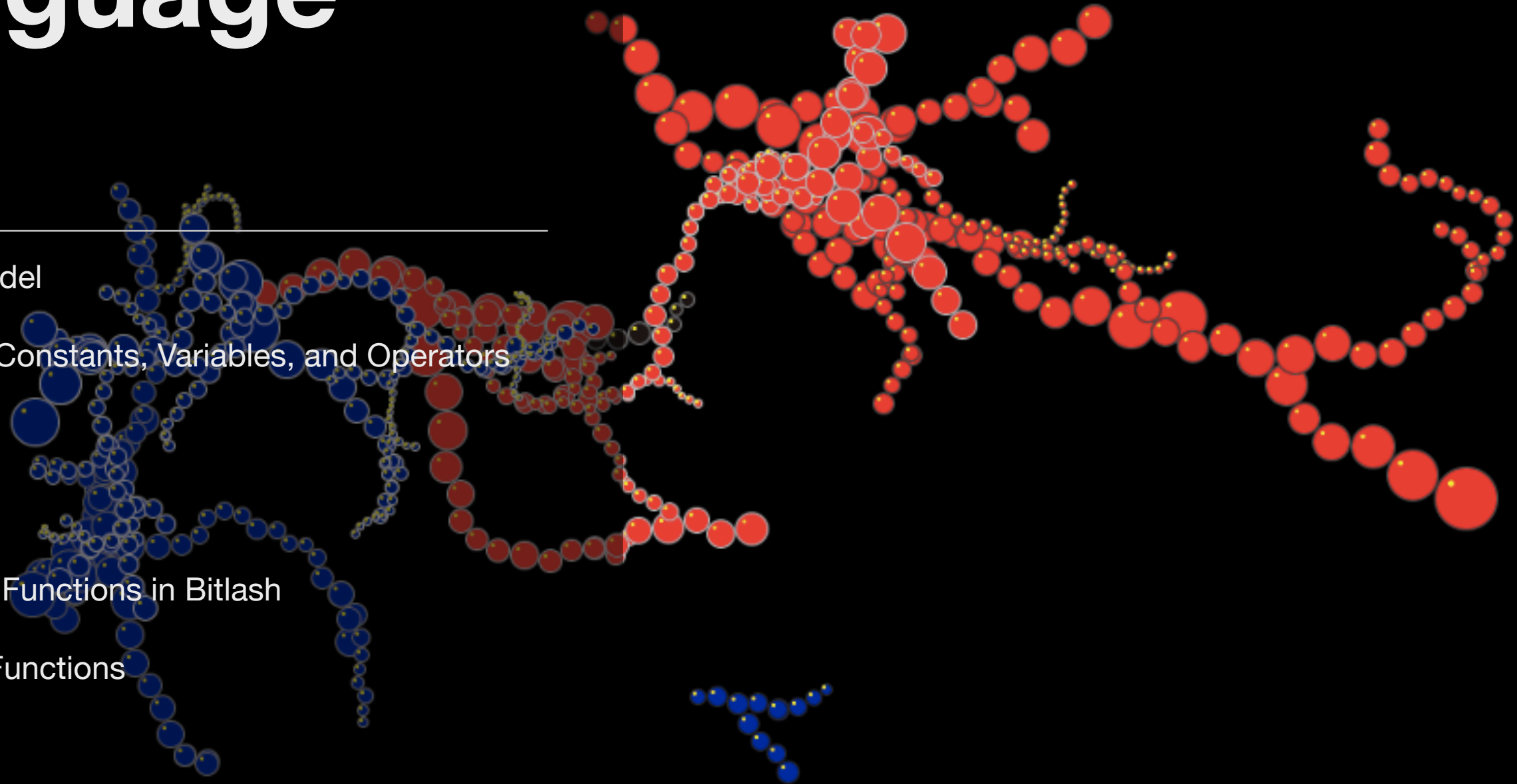
Language Overview

THE GRAMMAR

1. **A command line is one or more statements separated by semicolons**

2. **A statement is a command, an assignment, an expression, or a list of statements in {;}**

3. **Commands and Functions are described in the next sections**

4. **Conditionals (if, while, and switch) also get their own section**

5. **Expressions support variables a-z, 32-bit integer constants, fully recursive function calls with arguments, and C-style operators**

6. **String constants can be passed as arguments and used in the print command**

How It Works: The Bitlash Execution Model

Think of Bitlash as a dumb command line calculator. You type a line of commands and press Enter; Bitlash interprets the line and returns you to the prompt.

Think of a Bitlash function as a saved command line, with a name, stored in EEPROM. When you invoke to a function by using its name, Bitlash suspends what it's doing working your main command to execute the function, which is interpreted just as though you typed the function's text. Effectively it is a subroutine call.

Think of background functions as stored command lines scheduled to be run periodically. When you say **run toggle13,1000**, it means "whatever else may be happening, please execute the **toggle13** function about every 1000 milliseconds."

Numeric Constants

NUMERIC CONSTANTS

1. **Signed and unsigned 32-bit decimal constants: 123, -123**

2. **Hex constants: 0x0d**

3. **Char constants: 'x'**

4. **Binary constants: 0b10100101**

Decimal signed numeric constants in the range of a 32-bit signed integer are supported as you would expect.

Hex constants of the form 0xHHHHHHHH also work, as do single-quoted ascii character constants like 'q'.

Binary constants of the form 0b01010101 are also supported.

String Constants

STRING CONSTANTS

1. **Valid as function arguments and in the print command**

2. **Supports C-style backslash escape sequences**

String constants can be used as arguments and within the Print statement.

```
print "Hello, world";

printf("This is a string: %s", "some string");
```

A string standing alone is treated as a comment and ignored.

The rules for special characters in string constants are similar to C. The backslash character specifies that an escape sequence is to follow. Here are the supported escapes:

Char	Value	Description
\r	0xd	carriage return
\n	0xa	line feed
\t	0x9	tab
\\	'\'	a backslash
\"	""	a double quote
\xHH	HH	8-bit hex character HH

The hex form is especially useful in crafting escape sequences to print to peripheral devices.

Numeric Variables
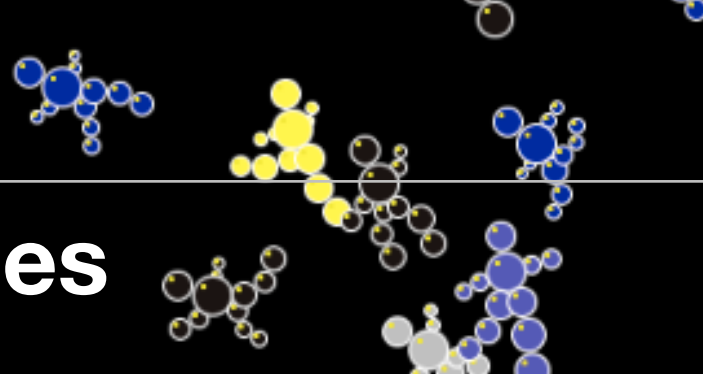
NUMERIC VARIABLES

1. **26 of them, named a through z**

2. **32 bit integer values**

3. **initialized to zero**

Bitlash provides 26 built-in 32-bit signed integer variables named lowercase 'a' through lowercase 'z' (though case is ignored).

Variables are initialized to zero at boot time.

```
> i=10; while i-- print i,
9876543210>
```

Pin Variables

PIN VARIABLES

1. **Special variables named D0 through D22 and A0 through A7 provide direct access to read and set IO pin values, e.g., D13=1**

2. **Shorthand for digitalRead/Write and analog Read/Write**

Pin Variables: Names for the Arduino Pins [a0..a7] and [d0..d22]

You can refer to specific analog and digital pins using **pin variables**. Pin variables are of the form a0, a1, a2, … a7 and d0, d1, d2, … d22.

When a pin variable is used in an expression it is a shorthand for digitalRead (for the d0.d22 pin variables) or analogRead (for a0..a7).

```
> x=d4            // digitalRead(4) and save it in x
> print x, a6     // print digitalRead(4) and analogRead(6)
```

Assigning to a digital pin variable does as you would expect: it turns the output on or off, just like digitalWrite.

```
> d13 = 1         // turn on D13
```

Assigning to the a-pin variables is a little tricky. You might think from the name that it does some sort of output to an analog pin. But, not so. Assigning to an analog pin variable does analogWrite, which is PWM output to a DIGITAL PIN!!

```
> pinmode(5, 1)   // set D5 to digital output mode
> a5 = 128        // generate 50% duty cycle PWM on pin 5
```

Operators

OPERATORS

1. **C arithmetic, relational, logical and bitwise operators are supported with customary precedence**

2. **Neither the compound assignment operators nor the member and pointer operators are supported (sorry, no +=, -=, <<=...)**

Here are the operators supported in Bitlash expressions, with meanings as in standard C; higher precedence binds more tightly.

Symbol	Name	Precedence
+	add	1
-	subtract	1
*	multiply	1
/	divide	1
()	parenthesis	1
<	less than	2
<=	less than or equal to	2
>	greater than	2
>=	greater than or equal to	2
==	equal to	2
!=	not equal to	2
<<	shift left	3
>>	shift right	3
^	exclusive or	4
&	logical and	4
|	logical or	4
++	pre- or post-increment	5
--	pre- or post-decrement	5

Commands

Here is an alphabetical reference list of Bitlash commands. (See also Bitlash functions.)

Expression evaluation

If you type a "naked expression" it will be evaluated, and any side effects like function calls and function executions will happen, but nothing is printed unless you say so using print. For example:

```
> d13=1; delay(125); d13=0
>
```

Your code is executed (d13 goes high for 125 ms), but all you see on the console is the command prompt after it's done.

boot: restart the arduino

Resets the Arduino. On restart, the **startup** function will run, if one is present.

function funcname { stmt; …; stmt;}

Defines a new Bitlash Function to be stored in EEPROM.

Bitlash functions have their own section.

```
> function blip {d13=!d13;}
> ls
function blip {d13=!d13;}
```

```
> rm blip
> ls
>
```

help: display some onboard help text

Help displays a short help message that can be helpful if you forget the name of a command or function. It also displays a list of your functions via 'ls'.

if (expr) {stmt; stmt;…} [else {stmt;stmt;...}]

The **if** command executes the first statement list if the test condition is true; otherwise, the optional else part is executed, if present.

```
> if (d4) {print "the light is on";}
```

ls

List all the functions stored in EEPROM.

```
> function blip {d13=!d13;}
> ls
function blip {d13=!d13};
```

peep: print a map of eeprom

Peep prints a map of eeprom usage. This can help you see how full your EEPROM is, and whether you have fragmented free space.

```
> ls
function bcn {printm{"vvv de w1aw");};
function zx {printm("00000000000000000000000000000");};
function toggle3 {d3=!d3};
function pt {printf("the time is %d", millis)};
> peep

E000:  bcn\ prin  tm{" vvv   de w 1aw"  );\. ....  .... ....  .... ....  ...z x\pr  intm ("00
E040:  0000 0000  0000 0000  0000 0000  000" );\t  oggl e3\d  3=!d 3\pt  \pri ntf(  "the  tim
E080:  e is  %d"  , mi llis  )\.. ....  .... ....  .... ....  .... ....  .... ....  .... ....
E0C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E100:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E140:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E180:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E1C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E200:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E240:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E280:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E2C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  ... .... ....
E300:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E340:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E380:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E3C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  ... .... ....  ....
>
```

Note that the "E000" address is actually EEPROM address 0000. Subtract 0xe000 from the addresses shown.

print [#pin:] [expr][,](expr][,) – print

In its simplest form: print foo,bar will get you started. But print has a lot of options, so please see the section on Printing for details.

ps: 'process status' – print a list of running background functions

Ps shows a list of running background functions.

```
> run t13
> ps
0: t13
```

rm: delete a function from eeprom

```
> function blip {d13=!d13;}
> ls
function blip {d13=!d13;}
> rm blip
> ls
>
```

Use "rm *" to erase the whole EEPROM.

run: run a function in the background

See the section on Background functions for details.

```
> run t13,125
> ps
0: t13
```

stop tasknum | stop | stop *

Stop a background task by number, stop the current task, or stop all tasks.

```
> run t13
> ps
0: t13
> stop 0
> ps
```

```
>
```

switch (expr) {stmt0; stmt1; …; stmtN;}

The switch command selects one of N statements to execute based on the value of a numeric expression. This provides a very easy way to build state machines, among other things.

Syntax:

```
switch (selection-expression) { stmt0; stmt1; ...;
stmtN;}
```

The selection-expression is evaluated to produce an integer result, which is used to select one of the supplied statements in curly braces. The selected statement, **and only the selected statement** from among those in curly braces, is executed, then execution continues after the switch statement.

(Note: This construct is similar to but not strictly compatible with the C switch statement, which has "case x:" tags to specify each case.)

If the value of the selection expression is less than zero it is treated as zero and therefore the first supplied statement is executed. If the value is greater than N, the last statement is executed.

Example: This switch statement in the elevator2.btl example calls **wating**, **goingup**, or **goingdown**, depending on the value of s, the variable that represents the state of the elevator system:

```
switch s {waiting; goingup; goingdown;}
```

If the value of s is <= 0, the **waiting** function will be called.

If the value of s is == 1, the **goingup** function will be called.

If the value of s is >= 2, the **goingdown** function will be called.

Example: Unlike Bitlash 1.0, the case handlers can be any Bitlash statement, for example:

```
switch (d) {print "foo"; print "bar"; print "baz";
boot;}
```

while (expr) {stmt1; …; stmtN;}

The **while** command repeats execution of the while-statements as long as the test expression is true. When the test expression is false, execution continues after the while's statement list.

```
> while (millis() < 100000) {d13=!d13;} print "Done";
(light flickers)
Done
>
```

Execution of the **while** loop is blocking. In other words, during the execution of a **while** command, no other functions can run, since Bitlash is working as hard as it can to get to the end of that command line. The combination of **while 1** with **delay()** is particularly deadly, since it uses the whole processor:

```
> print "this works but NOTE it will hog the whole ardu-
ino"
> while 1 {print millis(); delay(1000);}
```

```
(no prompt again until you press ^C)
```

Bitlash polls the serial port for Control-C during while loops, so you can break out.

Omitting { } for a single statement

When using **while** and **if**, it is legal syntax to omit the {} if only a single statement is to be executed, just like in C. (In other words a statement is recursively defined as a statement or a list of statements in curly braces.)

```
> i=0; while (++i<3) print i; print "done";
1
2
done
>
```

Nesting

Nested conditionals work as you would expect:

```
> a=0;while (a++<2) {b=0; while (b++<2) {print a,b;}}
1 1
1 2
2 1
2 2
>
```

How to simulate a "for" loop

The simplest way is to unroll it as a **while**. Here is an example that initializes pins 3 through 8 as outputs and sets them to HIGH:

```
> i=3; while i<=8 {pinmode(i,1); dw(i,1); i++;}
```

For extra credit, let's initialize the even ones on and the odd ones off:

```
> i=3; while i<=8 {pinmode(i,1); dw(i,(i&1)); i++;}
```

Built-in functions

BUILT-IN FUNCTIONS

1. **Many functions call Arduino or avr-glibc functions directly (so the names should be familiar)**

2. **For functions of zero arguments you may omit the empty parens, so millis and millis() are both valid**

This is an alphabetical reference listing all the built-in functions provided in Bitlash.

Many Bitlash functions are straight pass-throughs from the Arduino functions of the same name. Therefore, the definitive reference for the behavior of the functions is the Arduino extended functions reference at http://arduino.cc/en/Reference/extended.

Functions may be used in expressions in the normal way: abs(ar(3)–256) and so forth.

Bear in mind that functions return 32-bit integer values (**signed long** or **int32_t** in C). Whether the value is interpreted as signed or unsigned depends on the function.

It is an error to call a function with the wrong number of arguments.

For functions of zero arguments you may omit the empty parens ():

```
> print free,free(),millis,millis()
335 335 24455 24456
```

See also Bitlash commands and the Bitlash language.

abs(x): absolute value

Return x < 0 ? -x : x. See Arduino:abs at http://arduino.cc/en/Reference/abs.

ar(apin): analogRead(apin)

Return a 10-bit analog-to-digital conversion value from the specified **analog input pin**. See Arduino:analogread at http://arduino.cc/en/Reference/analogRead.

aw(dpin,value): analogWrite(dpin,value)

Write a PWM value to a ** digital pwm output pin**. The pin must be prepared for output beforehand via pinmode(pin,1).

A simpler syntax if the pin is fixed is: a6=128

See Arduino:analogwrite at http://arduino.cc/en/Reference/analogWrite.

baud(pin,baud): set baud rate for printed output

By default, Arduino prints at 57600 on pin 0 and 9600 on any other pin. If you wish to set a different rate use the baud function:

```
// prepare "print #5:" to produce serial output
// on pin 5 at 4800 baud (8,n,1)
> baud(5, 4800);

// set the default/hardware serial port to 9600 baud
> baud(0, 9600);
```

See printing.

bc(val, bitnum): bitclear

Returns val with the bit indicated by bitnum [0..31] set to 0.

See Arduino:bitclear at http://arduino.cc/en/Reference/bitClear.

beep(pin, frequencyhz, durationms)

Toggle the specified pin at the specified frequency for the specified duration. Automatically sets pinMode to OUTPUT.

Beep is blocking: background execution is paused. Use caution for long durations: there is no way to break out of a long beep (the longest value is several days).

```
// make the beeper on pin 11 beep at 440Hz for 200ms
> beep(11,440,200)
```

br(val, bitnum): bitread

Returns the value of the bitnum'th bit [0..31] in val.

See Arduino:bitread at http://arduino.cc/en/Reference/bitRead.

bs(val, bitnum): bitset

Returns val with the bitnum'th bit [0..31] set.

See Arduino:bitset at http://arduino.cc/en/Reference/bitSet.

bw(val, bitnum, bitval): bitwrite

Returns val with the bitnum'th bit [0..31] set to bitval [0!!0].

See Arduino:bitwrite at http://arduino.cc/en/Reference/bitWrite.

constrain(val,min,max)

Returns the closest value to val between min and max.

See Arduino:constrain at http://arduino.cc/en/Reference/constrain.

delay(milliseconds)

Pause execution for the specified number of milliseconds.

Delay is blocking; nothing else happens while a delay() is being processed. For this reason is it better to use background functions to do things that span non-trivial time.

See Arduino:delay at http://arduino.cc/en/Reference/delay.

dr(dpin): digitalRead(dpin)

The dr() function is shorthand for digitalRead(). It returns the current logic level on the specified pin.

If the pin is fixed and known beforehand you can use **pin variable** notation instead:

```
> z = 13
> x = dr(z)      // return digitalRead(13)

// same result using pin variable notation
> x = d13
```

See Arduino:digitalread at http://arduino.cc/en/Reference/digitalRead.

dw(dpin,bval)

digitalWrite: Set the designated pin to the given boolean value.

See Arduino:digitalwrite at http://arduino.cc/en/Reference/digitalWrite.

er(addr): EEPROM.read(addr)

Return the value stored in EEPROM at the specified address.

ew(addr, value): EEPROM.write(addr, value)

Write one-byte value to EEPROM at addr.

More on Bitlash and EEPROM at Bitlash Functions.

free()

Returns the amount of memory between the top of the heap and the stack pointer; in other words, the amount of stack space Bitlash and all your other code, including interrupt handlers, have to work with.

Numbers less than 50 or so indicate ram starvation and mean that the odd behavior you are seeing (or will see soon) is attributable to the occasional excursion of the stack into the defined ram area, piddling on the interpreter state. Nothing good will come of this.

Note that malloc() and free() are not used by or included in Bitlash and the free memory calculation takes no heed of memory broken out of the heap if you should include them for your code.

The free() function will issue an exception if free memory appears to be less than zero.

inb(reg)

Return the 8-bit value of the specified AVR processor register.

The processor registers and their functions are the function of the extensive ATmega328P Data Sheet at http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdf. You will find a convenient cross reference chart of the registers starting on page 425.

You specify a register address (from the first column of the table on pp. 425 ff.) to tell the inb() and outb() which register to address. In the case where the table has two addresses, use the right-hand one.

For example, to read the Timer0 count register TCNT0, find its entry on page 470 and note its address is 0x46. Then, to read and print it in Bitlash:

```
> print inb(0x46)
```

map(val, fromlow, fromhigh, tolow, tohigh)

Constrains val to [fromlow..fromhigh] and then maps it linearly to [tolow..tohigh]. Got it?

NOTE: The map() function is deprecated as of Bitlash version 1.1 and is therefore not included in the build.

See Arduino:map at http://arduino.cc/en/Reference/map.

max(a,b)

Returns the greater of a and b.

See Arduino:max at http://arduino.cc/en/Reference/max.

millis()

Returns the number of milliseconds since startup.

See Arduino:millis at http://arduino.cc/en/Reference/millis.

min(a,b)

Returns the lesser of a and b.

See Arduino:min at http://arduino.cc/en/Reference/min.

outb(reg,value)

Set the specified AVR processor register to the given value.

The processor registers and their functions are the function of the extensive ATmega328P Data Sheet at http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdf. You will find a convenient cross reference chart of the registers starting on page 425.

You specify a register address (from the first column of the table on pp. 425 ff.) to tell the inb() and outb() which register to address. In the case where the table has two addresses, use the right-hand one.

For example, to set the OCR0A register (0x47) to control timer 0 PWM, you could say:

```
> outb(0x47, 128)
```

pinmode(pin,outie)

Set the pin to an output pin if **outie** is true.

See Arduino:pinmode at http://arduino.cc/en/Reference/pinMode.

pulsein(pin, value, timeout)

Times an input pulse. See Arduino:pulsein at
http://arduino.cc/en/Reference/pulseIn.

random(max)

Returns a random number between 0 and max–1.

See Arduino:random at http://arduino.cc/en/Reference/random.

shiftout(dataPin, clkPin, bitOrder, value)

Bitbang the provided value out the data pin, clocking with the clock pin.

See Arduino:shiftout at http://arduino.cc/en/Reference/shiftOut.

sign(value)

Return –1 if value is negative, 0 if 0, or 1 if positive.

snooze(snoozems): defer background task execution for a while

Suspend further invocations of this background macro for snoozemillis
milliseconds. See the page on Background Functions for details on mul-
titasking with snooze.

```
> function blink13 {d13=!d13; snooze(100);}
```

```
> run blink13    // only toggles every 100 ms
```

Defining new functions

WRITING FUNCTIONS IN BITLASH

1. **To define a function: function newfunc {stmt;stmt;}**

2. **Functions are saved in EEPROM; use *peep* to see a map**

3. **To call a function: newfunc(22)**

4. **To run a function in the background: run newfunc, 1000 (no arguments)**

5. **To remove a function: rm newfunc**

6. **If a function named *startup* exists, it is run at boot time**

7. **If a function named *prompt* exists, it is run instead of displaying the '>' prompt**

Bitlash allows you to store and manage **Bitlash Functions** in the Arduino's EEPROM. By defining new functions, you extend Bitlash to suit your application.

Think of a function as a stored command line. In Bitlash, that can be multiple statements. And a statement within a function can call a function, so function execution can nest like subroutine calls (up to a dozen levels deep or so). But the simplest use is simply to give a name to a sequence of commands.

Rules For Function Names

Function names must begin with an alpha character 'a'..'z' and be 12 characters or less in length. Function names must be more than one character long (to avoid conflict with the built-in numeric variables named a through z) and must not conflict with any of the pin variables (d0..22 and a0..7), nor with the built-in functions and commands. Function names can contain digits and the underscore '_' and period '.' characters.

Defining a Function

To define a function, use the **function name {stmt;stmt;}** syntax:

```
> function toggle13 {d13=!d13;delay(100);}  \\ defines toggle13: flip
the bit and wait a while
> function hello {print("Hello, world!);}
```

Run a function: foreground

To invoke a function, you can type its name at the command line or refer to it in another function (for example in the definition of startup above):

```
> toggle13          \\ flips bit 13
```

Run a function: background

To run a function in the background use the **run** command:

```
> run toggle13
> ps              \\ is anything running?
0: toggle13       \\ yes, there's our function running
> stop 0          \\ stop it
>                 \\ now it's stopped
```

There is more detail on this topic in the Background Functions section.

List functions in EEPROM

To list all the functions stored in the EEPROM use the **ls** command:

```
\\ list all the functions
> ls
function toggle13 {d13=!d13;}
function hello {print("Hello, world!";}
```

Remove a function from EEPROM

To remove a function use the **rm** command:

```
\\ rm removes a function.  rm * does what you think.
> rm hello
> ls
function toggle13 {d13=!d13;}
```

The Startup Function

If there is a function by the name "startup", it is run automatically at boot time. Make one like this:

```
> function startup {print 1,2,3;}
> boot
bitlash v2.0 here!...
1 2 3
>
```

Don't like it?

```
\\ redefine the startup function
> function startup {...new definition...}

\\ delete the startup function
> rm startup
```

Tip: If it looks like it's sitting there doing nothing, it's probably running a function. Press ^C to break out of a looping startup command.

The Prompt Function

If there is a function by the name "prompt", it is run automatically whenever the command line prompt is to be printed so that you can customize the prompt. Here is an example of a prompt function that prints the current time in millis:

```
> function prompt {print millis,;print "$",;}
36244$        \\ press enter at this prompt
36484$        \\ time advances
```

Inspecting and Formatting Function Storage in the EEPROM

There may be debris in your EEPROM from another project. Or your Bitlash program can blow chunks, or Bitlash can blow chunks. Anyway, the EEPROM can become "less than fresh". This might first show up as funky results from the ls command, for example.

You can inspect your EEPROM with the peep command, which prints a map of EEPROM:

```
> ls
function bcn {printm{"vvv de w1aw");};
function zx {printm("00000000000000000000000000000");};
function toggle3 {d3=!d3};
function pt {printf("the time is %d", millis)};
> peep

E000:  bcn\ prin  tm{" vvv   de w 1aw"  );\. ....  .... ....  .... ....  ...z x\pr  intm ("00
E040:  0000 0000  0000 0000  0000 0000  000" );\t  oggl e3\d  3=!d 3\pt  \pri ntf(  "the  tim
E080:  e is  %d"  , mi llis  )\.. ....  .... ....  .... ....  .... ....  .... ....  .... ....
E0C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E100:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E140:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E180:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E1C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E200:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E240:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E280:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E2C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E300:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E340:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E380:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
E3C0:  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....  .... ....
>
```

This is a healthy map. The places marked '.' are empty (==255). The name-value storage for toggle13 and bcn can be seen. As you add functions you will see them fill from low addresses up, always in pairs.

An unhealthy map might have garbage in the supposedly unused part. Or there could be free space available but spread around in fragmented blocks (see Fragmentation below).

You can erase and "reformat" the EEPROM using the "rm *" command; see below. This will erase any functions you have typed into bitlash, as well as the garbage. In other words:

Note: RM *" WILL NUKE THE WHOLE EEPROM. There is no way to recover it. Please use caution.

A note on EEPROM usage: Don't bang on the EEPROM

The EEPROM is said to be certified for about 1e5 cycles. Bitlash could drive that many write cycles in under ten minutes, if you told it to. If you do this and your EEPROM breaks you get to keep the pieces. But please write and let us know how it failed. We've never seen it happen. Anyway, use EEPROM for long term storage like function definitions, not loop counters, and factor EEPROM life into your application life cycle model.

EEPROM Free Space Fragmentation

Heavy use of the function store may lead to fragmented free space. You would see this in the peep map as free space dot clusters too small to be of use scattered here and there.

This version of Bitlash does not have a method to compact the free space, but if you are highly motivated to squeeze out the last possible byte, here is a straightforward but unfortunately manual workaround:

- Use 'ls' to get the contents of all the functions. Copy to safe place!

- Use 'rm *' to nuke the eeprom. All your functions are gone! Hope you copied them!

- Paste the output you saved into your bitlash terminal emulator to re-define the functions.

Depending on many factors, (baud rate, clock speed, terminal program, OS, …) it may be necessary to paste the definitions one at a time.

Reserving EEPROM for Other Applications

A portion of the EEPROM space can be kept from bitlash for use by your application's C code by adjusting the values of STARTDB and ENDDB in bitlash.h. You are advised to save, reformat, and reload any EEPROM contents after changing these values.

Say you want to reserve 32 bytes at the high end of EEPROM (you'll be using EEPROM.read() and EEPROM.write to manipulate it), edit the #define of ENDDB to be

```
#define ENDDB (E2END – 32)
```
Bitlash will then use memory from STARTDB to ENDDB - 1. Your 32 bytes run from ENDDB to ENDEEPROM - 1. To read byte 15 of your private EEPROM, use EEPROM.read(ENDDB + 15).

You can use the bitlash "peep" command to dump the EEPROM, which will include your space. Telling bitlash to "rm *" will delete all of your bitlash functions from EEPROM, but will leave your private space unmolested.

It's up to you to manage your private memory, including providing a way to clear it, etc. One reasonable approach is to add a new bitlash C command or two to interact with and manipulate your private EEPROM space.

Note that flashing your firmware from Arduino will NOT erase your reserved EEPROM. (This is a feature, not a bug.)

Background Functions

BACKGROUND FUNCTIONS

1. **You can start up to 10 tasks to be periodically run in the background**

2. **Any function-of-no-arguments can be a task**

3. **To start a task: run beacon, 250**

4. **To list tasks: ps**

5. **To stop a task: stop 3**

6. **To stop all tasks: stop * or Control+C**

7. **To pause all background tasks: Control+B**

8. **In a task, to schedule the next task time, call snooze(time-in-ms)**

Bitlash can run up to 10 functions in the background while you work in the foreground at the command prompt.

The Run Command

The **run** command runs a function in the background:

```
> run toggle13  // LED on D13 starts flashing
>    // and you get the foreground prompt back
```

Think of the run X command as being a "while 1 X" that runs in the time between your key-strokes.

In the above example, toggle13 will be called Very Frequently, perhaps as often as once each time your loop() procedure calls runBitlash().

Bitlash stops all background functions when you press ^C. You can also stop a specific one using ps (to discover its process id) and stop.

```
> ps             // list running background functions
0: toggle13      // toggle13 is number 0
> stop 0         // or stop * or ^C to stop all
>
```

Run Command: Optional [,snoozems] argument and changes to snooze()

The "run" command has an optional **snoozems** argument to provide control over how often each background task is run. This is convenient for tasks that run on a fixed timer: many functions can dispense with the snoozing thing entirely if the runtime interval is known when the function is started.

For example, this will run the function t1 every 27 millis or so:

```
run t1,27
```

If the optional **snoozems** argument is not specified it is treated as zero: in other words, you must manage the snooze interval in your function's code; see below.

This change also has a small impact on semantics of the snooze() function. Previously, snooze() would set the time-at-which-the-task-is-eligible to run immediately when called.

The new behavior is subtly different: the scheduling takes place somewhat later, when the function exits the current invocation, not at the time of some call to snooze(). Bitlash maintains a snoozems value for each background task; by default it is zero, and the task is called as fast as the round-robin gets back to it. If you specify a value in the run command or by calling snooze, this value is saved and applied in the rescheduling calculation when the task exits.

Run Your Application Automatically in the Background

Use the startup function to do a run on your top-level function and your application can run in the background while you have control of the keyboard to poke at it:

```
> function startup {run myapp}
> boot
bitlash here! v2.0...
> ps
0:myapp
```

Don't delay(); use snooze() instead

If your objective is to present the illusion of multitasking, you should avoid using the delay() function in functions, since everything else comes to a screeching halt while delay() is happening.

Foreground keyboard entry may become sluggish if you have many tasks with delay()s more than a few millis each.

Bitlash provides the snooze(ms) function to give your background function tasks a way to delay without hogging the CPU. Bitlash suspends a task which calls snooze() until the specified number of millis has passed.

NOTE: Calling snooze() has no apparent immediate effect. Your function continues to execute, if there is further code after the snooze(). But bitlash will not re-enter your task until the specified time has passed.

Here is an improved toggle13 that uses snooze() instead of delay():

```
> function snooze13 {d13=!d13; snooze(100);}
> run snooze13
```

>

Since snooze() means "don't call me again for a while" instead of "please spin away some time", you will find that your foreground typing and responsiveness of other background functions is much improved compared with "run toggle13".

Example: Asynchronous Background Tasks

Here is an example of three asynchronous background tasks each doing its thing on its own little timeline, using snooze() to set its calling interval:

```
> function chirp {print "chirp ",; snooze(2500);}
> function eep {print "eep ",; snooze(800);}
> function ribbit {print "ribbit ",; snooze(3500);}
> function nightfall {run eep;run chirp;run ribbit;ps}
> nightfall
0:eep
1:chirp
2:ribbit
> eep chirp ribbit eep eep eep chirp eep
ribbit eep eep chirp eep eep ribbit eep
chirp eep ^C
```

Printing

PRINTING TIPS

1. **For simple printing use the print command:**
   print "hello, world", 1, 2, 3;

2. **A C-like printf() function is also built-in:**
   printf("%3d", 7);

3. **Print to any IO pin at your choice of baud rate**

4. **See the examples/ folder for code to customize the output handler**

Printing Overview

There are two ways to print in Bitlash: the legacy **print** command, and the **printf()** function. The **print** command is handy for printing basic values without much formatting control; it was part of Bitlash 1.0 and is partly retained mainly for backwards compatibility. The ":x" format specifier language will be deprecated in Bitlash 2.1, leaving only the printing of basic values.

The **printf()** function is new in Bitlash 2.0. It is more compatible with standard C and much more capable. It will be the supported printing pathway going forward.

printf("format string", value1, value2,… valueN);

The printf() function prints values to the console.

The values are printed under control of a format string, which must contain a specifier for each value to be printed.

Specifiers are of the format:

```
Format Specifier := <percent-sign> [<pad-char>] [<width>](<pad-char>]
[<width>) <specifier>
Example: printf("%d", 7)    prints "7"
Example: printf("%3d", 7)   prints "  7"
```

```
Example: printf("%03d", 7)      prints "007"
Example: printf("%s","Hello")   prints "Hello"
Example: printf("%%")           prints "%"
```

Table of printf() Format Specifiers

Specifier	Print Format	Pad
d	Decimal, signed	blank unless '0' is specified
x	Hexadecimal	blank unless '0' is specified
u	Decimal, unsigned	blank unless '0' is specified
b	Binary	blank unless '0' is specified
specifier	String	blank
c	Character	the char
#	Print-to-pin	pin number
%	Print '%'	

Special Characters

The percent sign signals a format specifier in the string. To print a percent sign, you must double it:

```
printf("%%")     prints '%'
```

The printf() function does not automatically provide a newline at the end of printed output. You must indicate line endings using the '\n' character:

```
printf("The time is: %d\n", millis);
```
You can specify arbitrary hex values in a string constant using \xHH notation:

```
printf("\x30");     prints '0'
```

Here is a table of the special characters supported by Bitlash in string constants:

Char	Value	Description
\r	0xd	carriage return
\n	0xa	line feed
\t	0x9	tab
\\	'\'	a backslash
\"	'"'	a double quote
\xHH	HH	8-bit hex character HH

Padding

Normally, numeric values and strings are pre-padded with blanks to fill out the specified field width. You can pre-pad numeric values with zeroes instead by adding a zero before the width character:

```
printf("%03d", 7)    prints "007"
```

The "*" character used as a width

If the special character '*' is used in place of a width specifier, the next value in the passed-in value list is used as the width:

```
printf("%*d", 3, 7)      prints "  7" - padded to 3 char-
acters with ' '
```

Printing to Alternate Serial Pins

You can redirect output to a non-standard serial pin using the '#' specifier, which picks up the pin number to use from the next value in the value list:

```
// prints "7" to pin 4 at the default 9600 baud
printf("%#d", 4, 7)
```

It's a kludge, but the width specifier can be used to override the default 9600 baud sending speed:

```
// prints "7" to pin 4 at 4800 baud
printf("%4800#d", 4, 7)
```

Specifying the baud rate in this way has the same effect as calling baud(4,4800): the pin will remain at the indicated baud rate until it is changed again.

ACCESSING SECONDARY UARTS WITH PRINTF

If you are using an Arduino Mega 1280 or Mega 2560 with Bitlash and you print to pin 18 then Bitlash will automatically use the Serial1 UART rather than software serial for I/O. (There is currently no support in Bitlash for accessing Serial2 or Serial3 from printf.)

Likewise if you are using a Sanguino with the AVR ATmega 644P processor, I/O to pin 11 will use the second UART of the Sanguino rather than software serial.

The Print Comand

This section documents the legacy Print command and its options.

The print command will be retained in Bitlash going forward, but the ":x" sublanguage will be deprecated in a future version. Users are advised to migrate to the printf() function with all due haste.

Basic Printing: a list of items

The simplest form of the print command causes bitlash to print out a bunch of numeric values separated by spaces, followed by a newline:

```
> print 1, 2+2, 1<<3
1 4 8
>
```

The print command can print string constants as well as numeric expressions, as you may recall from the Hello World example:

```
> print "Hello, world!", 123
Hello, world! 123
```

To suppress the blank and automatic end of line, print a single value followed by the comma, like this:

```
> print "Time:",      // Prints Time: with no trailing
blank or newline
```

Printing special characters

The rules for special characters are similar to C. See Language for details on constructing string constants with arbitrary ascii characters.

Hex, binary, and raw byte output

You can specify a modifier to produce several different output types by appending ":spec" to any expression in a print statement. Perhaps an example is quickest:

```
> print 33, 33:x, 33:b, 33:y, 33:*
33 21 100001 ! ******************************
```

Here is a table mapping the Arduino equivalents for hex, binary, and raw byte output:

to print	use :code	example	prints
hex	:x	print 32:x	20
ascii byte	:y	print 32:y	ascii space
binary	:b	print 32:b	10000
bars	:*	print 3:*	***

Printing Bars

You can print a bar of N '*' characters using the : format specifier, as shown in the example above. Any single-byte symbol will work (+-*/<>%:). Here's a cheap, low-res oscilloscope on pin A3 with adjustable temporal resolution:

```
> while 1: print map(a3,0,1024,10,50):*; delay(5)
```

Serial Output to Alternate Pins: print #n:…

Bitlash includes a software serial output capability which supports transmitting on any available output pin at a user-specified baud rate. This makes it straightforward to integrate additional serial devices with an Arduino.

You send output to an alternate output pin using the "print #n:" construct, where "n" is the Arduino pin number designator for the desired the output pin:

```
> print #6:"EMERGENCY DESTRUCT SEQUENCE ALPHA CONFIRMED"
```

By default bitlash will send output at 9600 baud. If you require a different baud rate use the baud function:

```
> baud(4,4800)      // set baud on pin 4 to 4800
> print #4:"ATZ"    // send some stuff on pin 4
```

Note: print #n: is blocking: background tasks will pause while output is being printed.

EXAMPLE: CLEAR THE SCREEN ON SPARKFUN SERIAL LCD

Here is an example combining the use of printing to an arbitrary pin, printing special characters, and (importantly!) suppressing the inter-item and inter-line spacing that bitlash would normally generate. Assume the serial LCD is on pin 4 and that all pin setup and baud rate initialization has been performed. Then this macro will clear the screen:
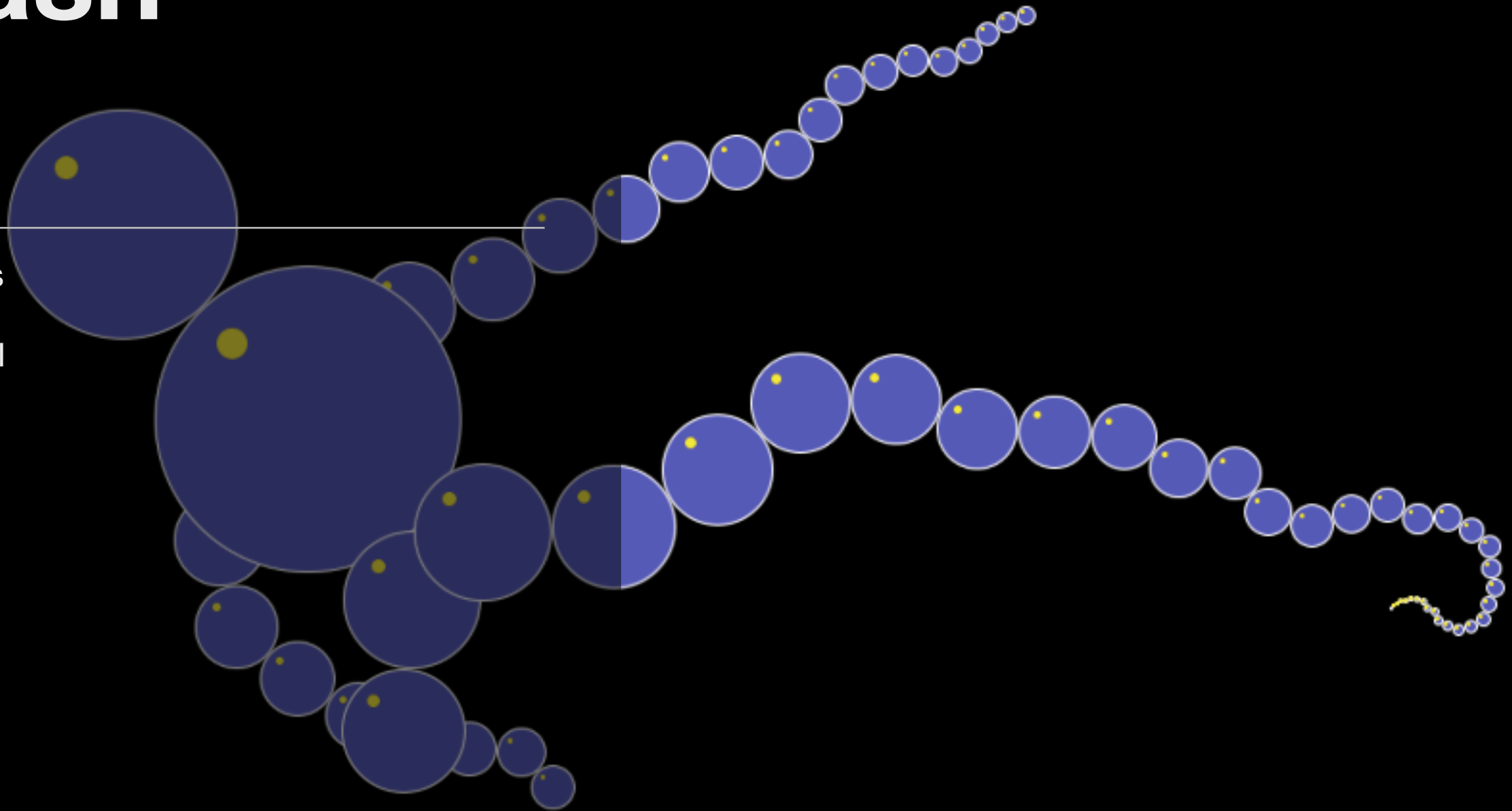
```
function cls {print #4:"\xfe\x01\",;}
```

Extending Bitlash

User Functions

The Bitlash API

User Functions

USER FUNCTIONS

1. **Copy from an example! See the examples/ folder**

2. **addBitlashFunction() registers a new function handler with Bitlash**

3. **Your function must be declared as: numvar newfunc(void)**

4. **In your C fuction, the passed-in argument count is getarg(0) and the arguments are getarg(1)..getarg(N)**

While Bitlash functions are a handy and straightforward way to extend Bitlash, there are cases where you need native C code to get closer to the hardware, perhaps for speed or to interface with a custom peripheral.

Bitlash provides an easy way to call C functions from Bitlash code, just like the Bitlash built-in functions. In effect, you add your functions to the Bitlash function library, and they become callable from the Bitlash command line.

The work happens in a function named **addBitlashFunction()** that is exposed as part of the Bitlash API. You call **addBitlashFunction()** in your sketch to register your C function with Bitlash and associate it with a function name. Then Bitlash calls your C function when it is invoked from the command line (or a function…).

Bitlash 2 introduces a new variable-argument scheme, using the getarg() function to fetch passed arguments from the interpreter core. Users of earlier versions will want to carefully review the section on fetching passed arguments below.

If you follow a few rules for setting up your function, you can wrap it in Bitlash without having to write a line of parsing code. You get all the parsing, functions, printing, and the rest of the runtime library for free.

Which is why it is worth reading the next section to learn how to set up your C function so Bitlash can call it properly.

Declaring the C function

Bitlash exposes the C variable type named **numvar** representing (in the Arduino version) the "signed long" integer data types. All internal calculations are done using fixed-point arithmetic in this precision.

So the first rule is that your function must return a value of type **numvar**.

Here is a simple example to walk through the plumbing. This example code is in the examples folder supplied with Bitlash, so you can open it in the Arduino IDE by selecting File -> Examples -> bitlash -> userfunctions if you'd like to follow along.

Suppose for some reason we need to be able to grab the count value from the AVR's internal Timer1 timer and use it in Bitlash.

In the Arduino sketch, we declare a user function named "timer1", taking no arguments, returning a numeric Bitlash value containing the timer count register value we're after:

```
numvar timer1(void) {
return TCNT1;    // return the value of Timer 1
}
...void setup(void)...
```

This completes the definition of the user function. What remains is to call **addBitlashFunction()** to register this new handler.

Registering a Function With addBitlashFunction()

You call **addBitlashFunction()** from your C startup code to register your function with Bitlash.

addBitlashFunction() takes two arguments: * a name for the new function * the function handler to call with "(bitlash_function)" cast before it

While you can call **addBitlashFunction()** any time, one would normally call it from setup(), like this, to complete the code for the example:

```
void setup(void) {
    initBitlash(57600);      // must be first to initial-
ize serial port

    // Register the extension function with Bitlash
    //      "timer1" is the Bitlash name for the func-
tion
    //      (bitlash_function) timer1 tells Bitlash
where our handler lives
    //
    addBitlashFunction("timer1", (bitlash_function)
timer1);
}

void loop(void) {
    runBitlash();
}
```

The setup() function initializes Bitlash and then calls **addBitlashFunction()** to install timer1(). The loop() function runs Bitlash as usual.

Now when Bitlash starts up it knows a new word:

```
bitlash here! ...
```

```
> print timer1(), timer1(), timer1()
22 194 67
```

The Function Name

A function name is a string constant of up to IDLEN (11) characters, like "timer1". It must start with a letter, and may contain letters, numbers, and the underscore '_' and period '.' characters.

Overriding the built-in function names is not supported; the user functions are searched last so if you use a built-in name your function will never be called.

Fetching Passed Arguments with getarg()

Your bitlash_function is declared (void). How does it get argument values from Bitlash?

Starting with Bitlash 2.0, a function does not need to accept any particular number of arguments. The arguments being passed to your macro are available to your code as via the Bitlash function API point getarg().

The value returned by getarg(0) is the count of arguments passed to your function. You can fetch the value of those arguments using getarg(1), getarg(2),… and so on to getarg(getarg(0)).

See the file examples/tonedemo.pde for an example of how to use getarg(0) to control how your function processes a variable number of input arguments.

The Function Handler Argument

The function argument to **addBitlashFunction()** provides Bitlash with the name of the C function handler you wrote to handle your user function. A small syntactic complexity: It is required to cast your function to the type "(bitlash_function)" to pass it to Bitlash, which is why the mantatory text "(bitlash_function)" appears before the function name in the example:

```
...
addBitlashFunction("timer1", (bitlash_function) timer1);
...
```

If you omit the cast the compiler will complain about function type mismatches.

NOTE: Maximum Number of User Functions

Bitlash ships with MAX_USER_FUNCTIONS set to 10 and will throw an exception if you try to install more.

If you need more functions, adjust the definition of MAX_USER_FUNCTIONS in bitlash-functions.c.

Coding User Functions

Here are some notes on limitations and issues you may need to address when coding a user function.

Sharing the CPU

Calling delay() from a user function is generally to be avoided, since delay() blocks everything else from happening.

Bitlash background functions don't (well, can't) run while your function is running, since they get time on a poll from loop(). This may affect your design.

Calling Internal Bitlash Functions

Generally it's fair game to call any internal Bitlash functions you want from your user function; you're inside the interpreter, after all. But there are things you should be cautious about in addition to the CPU time issue discussed above.

The interpreter is not re-entrant, and is halfway through parsing and executing a command line when it encounters your function reference. Don't re-enter Bitlash by calling **doCommand** or **runBitlash** from your user function.

Avoid recursion and stack-local storage where possible in user functions. Bear in mind that your function is executing inside the parser and is competing with the parser for stack space. Deeply nested recursion from a user function can cause a stack overflow, and local variables in your function only make this happen faster.

You can call the get_free_memory() function (which handles the Bitlash free() function) to see how much stack space is available when your function runs.

Unfortunately, it is very difficult to provide hard and fast rules about how much stack will be required for your application; experimentation is the best way to be sure. Personally, I get uncomfortable when free() goes below 200 but I have seen applications that require much more.

Printing From User Functions: Console Redirection

Printing from a user function using the Arduino Serial.print() function works as usual: it goes to the console as you would expect.

You can also print through Bitlash using the Bitlash printing primitives: sp(), spb(), and speol(), about which you will find more in bitlash-serial.c. If you use the Bitlash printing primitives, you will gain the benefit that any serial output redirection that is in effect when your function is called will be applied to your printed output. In other words, your function can automatically print to whatever pin Bitlash has selected as the output pin.

Here's an example to clarify. Suppose we have these two user functions wired up as **sayhi1** and **sayhi2**:

```
void sayhi1(void) { Serial.print("Hello"); }
void sayhi2(void) { sp("Hi"); }
Consider this bitlash code:

print #3: sayhi1()
print #3: sayhi2()
print #4: sayhi2()
```

The first line will print "Hello" to the serial console. Serial.print() output always goes there.

The second Bitlash command will print "Hi" to the device on pin 3, since "redirection to pin 3" is in effect when the function is called.

Likewise, the third command will print "Hi" on pin 4; thus a single user function can drive multiple output streams without needing special code.

Examples

See the examples folder in the Bitlash distribution for more. You can load examples in the Arduino IDE using the File -> Examples -> bitlash menu.

The Bitlash API

THE BITLASH API

1. In setup(): initBitlash(baudrate);

2. in loop(): runBitlash();

3. It's C. Call anything you like. See src/bitlash.h

4. A few useful API entry points are documented in this section

Your Arduino C program can interact with Bitlash using the functions documented here. Of course, in the tiny and open world of Arduino most everything is visible globally. So feel free to dive in and call what you need. The entry points documented here are intended to be reasonably well hardened for third party use.

initBitlash(baudrate);

You must call this first, normally from setup(), to initialize Bitlash and set the serial port baud rate.

runBitlash();

You must call runBitlash() from your loop() function to make Bitlash go. The more frequently you call runBitlash(), the more smoothly foreground and background activity will run. If you don't call runBitlash(), nothing will happen.

Example: Simple Integration

This is the minimum possible integration, and in fact it is the bones of the code that you will see at the bottom of bitlashdemo.pde:

```
void setup(void) {
    initBitlash(57600);
}
void loop(void) {
    runBitlash();
```

```
    // YourOtherCodeHere();
}
```

doCommand(char *command);

A simple way to control Bitlash is to use the doCommand() function to execute commands from your code. Your code can call doCommand() to make Bitlash "do stuff" – anything you can type, actually, up to the buffer size limit. Here's a dumb example that leads up to our next big case study: this code spends a lot of energy looking for the precise millisecond to beep at the top of the hour:

```
void setup(void) {
    initBitlash(57600);
}

void loop(void) {
    runBitlash();
    // beep at the top of the elapsed hour
    if (millis() % (60*60*1000) == 0) {
        doCommand("beep(11,440,1000)");
    }
}
```

Lots of ways to do this better (read on for one example), but the takeaway point is that your C code can drive Bitlash.

doCharacter(char c)

It is also possible to drive Bitlash's internal command line editor one character at a time. This is convenient if you have a character input device.

Bitlash will buffer and echo each character until you send a carriage return '\r' or linefeed, whereupon it will execute the line, the same way it works from the console.

See the examples/bitlashtelnet2.pde telnet server for a sample integration using doCharacter().

Console redirection with setOutputHandler()

You can arrange to capture all the output that Bitlash would normally send to the console serial port, and redirect it to suit your application. Use the setOutputHandler() api to direct Bitlash to route its output to your character-handling function, which should be defined as taking a single byte and returning a void, as in:

```
setup() {
    ...
    setOutputHandler(&serialHandler);
}

void serialHandler(byte b) {
    ...
}
```

There are four examples in the Bitlash distribution that show how this works:

See examples/loutbitlash.pde for a silly example that optionally makes all the output uppercase (therefore loud).

See examples/bitlashtelnet.pde for an example implementing both input injection and output capture. examples/bitlashtelnet2.pde is similar but uses the doCharacter() api instead of doCommand().

And of course the Bitlash web server, BitlashWebServer, uses this console redirection capability. See examples/BitlashWebServer.pde

User Functions: addBitlashFunction() and getArg()

You can extend Bitlash at compile time by adding "C User Functions" to the built-in function library. See the [User Functionsluserfunctions) page for details.

Manipulating Bitlash Variables from C

Bitlash has 26 variables named a through z. These are 32-bit signed integer values. Your code can read and write these values using a few simple functions. This allows your C code to control code running in Bitlash, and vice versa.

You refer to a variable using an integer index in the range [0..25] corresponding to 'a'..'z'. For example, here is how your code could use the getVar command to read out the value of the bitlash variable 't' and put it into the signed long C variable named 'temperature':

```
void loop(void) {
...
long temperature = getVar('t'-'a');
...
```

Here are the three API functions for Bitlash variable manipulation. Examples of their use are shown below in the clock application.

assignVar(char var, long value);

Assigns the (signed long 32-bit integer) value to the indicated Bitlash variable.

```
assignVar(0,42);              // a=42

// notation to calculate the var index:
assignVar('i'-'a', 33);       // i=33
```

long x = getVar(char var);

Returns the value of the given Bitlash variable.

incVar(var);

Increments the designated Bitlash variable.

```
incVar('i'-'a');              // i++
```