The engines under the hood:

This is for background information only and summarises the engines that are (or will be) under the hood (it is not part of the public Arduino `Print` API, but might help understanding the implementation options.)

```
unsigned int printNumber  (unsigned long number, unsigned int formatControl);
unsigned int printDouble  (double number,        unsigned int formatControl);
unsigned int printExponent(double number,        unsigned int formatControl);    // *)
unsigned int printChars   (char* string,         unsigned int formatControl,
                                                 unsigned int length=0     );    // *)
// *) to be defined and implemented
```

The format control is based on two bytes packed with the following information:

```
Field       range    bits  // description
size        1...32   5
strict      bool     1     // size specification also gives maximum field size  *)
fillZero    bool     1     // use zero for padding
leftAdjust  bool     1     // pad field on the right side
signed      bool     1     // item is signed (internal usage only)
forceSign   bool     1     // always output sign
spare       bool     1     // reserved (possibly to mark negated option)
radix       1...32   5     // radix (only for printNumber)
exponent    bool     1     // use exponent (only for printDouble / printFloat)
precision   0...15   4     // (only for printDouble / printFloat)

// *) to be defined and implemented
// to be foreseen: char* specific format control
```

`PrintFormat` Public API

The two bytes of format control are encapsulated in a `PrintFormat` class. This class and various basic predefined instances of `PrintFormat` are part of a public `PrintFormat` API.

```
const PrintFormat SIZE(unsigned char size);
const PrintFormat PRECISION(unsigned char precision);
const PrintFormat RADIX(unsigned char radix);
const PrintFormat BIN = RADIX( 2);
const PrintFormat OCT = RADIX( 8);
const PrintFormat DEC = RADIX(10);
const PrintFormat HEX = RADIX(16);
const PrintFormat STRICT;
const PrintFormat FILLZERO;
const PrintFormat LEFTADJUST;
const PrintFormat FORCESIGN;
```

These basic predefined instances can be combined into more complete `PrintFormat` specifications. Examples of various possible options are given below for discussion, including some more exotic options brought up in the discussion on the Arduino developers group:

```
PrintFormat myFormat = SIZE(6) ⊕  HEX ⊕  FILLZEROS;  // with ⊕ in { +, |, ||, &, &&, %, ...}
PrintFormat myFormat = PrintFormat(SIZE(6) ,  HEX ,  FILLZEROS);
PrintFormat myFormat = PrintFormat().asSIZE(6).asHEX().asFILLZEROS();
```

Note that the right hand side of the equals sign can also be passed directly as the second parameter of the print method.

```
serial.print( 1, SIZE(6) ⊕  HEX ⊕  FILLZEROS );  // with ⊕ in { +, |, ||, &, &&, %, ...}
serial.print( 2, PrintFormat(SIZE(6), HEX, FILLZEROS) );
serial.print( 3, PrintFormat().asSIZE(6).asHEX().asFILLZEROS() );
```

Discussion on combining format control:

All the proposed combination operators and methods fulfil the requirement that any combination based on compile time constant `PrintFormat` instances, can be evaluated at compile time. Actually, all options might be implemented in parallel (which does not mean that I would favour such an approach).

I have left out the 'naked' comma option:

```
serial.print(42,SIZE(6), HEX, FILLZEROS)
```

This option is confusing, not only for the user, also for the compiler and the implementation.

Amongst the operator overloaded `operator` ⊕ with ⊕ in `{ +, |, ||, &, &&, %, ...}`

`operator+`	does not seem natural i.e. : `HEX + HEX = HEX` ??
`operator|`	mostly reflect what is going on under the hood and fulfils `HEX | HEX = HEX`, however, semantically it may be weaker, the user wants to use `SIZE(6)` and `HEX` and `FILLZEROS`
`operator&`	seems more natural to me, also `HEX & HEX = HEX`. Semantically it reflects the user's intention to use `SIZE(6) & HEX & FILLZEROS` (logically it fits if you think in terms of negative logic). Note that `SIZE(6) & SIZE(8)` (which equals `SIZE(8)`) is still confusing.
`operator%`	just give me a good reason.

`PrintFormat(SIZE(6), HEX, FILLZEROS)` is ok for me.

`PrintFormat().asSIZE(6).asHEX().asFILLZERO()` I have not yet studied the Arduino style guide yet, but I doubt if this is Arduino like…, however on popular demand it can be implemented.

Question: is there any use for a negating `PrintFormat` option:

```
const PrintFormat myFormat = SIZE(6) & HEX & FILLZEROS;
Serial.print(42, myFormat & ~FILLZEROS); // no leading zeros in the answer to this question !!
```

Discussion on name conflicts

The name of the class `PrintFormat` could be simplified to `Format`, however I fear that there might be a clash with existing user code. We also might consider making `Format` part of a `namespace _Print_ {}`, however this keep long names: `_Print_::Format`, unless using '`using namespace _Print_;`' is sufficiently Arduino like.

The same discussion holds for the `PrintFormat` primitives `SIZE(), HEX, FILLZEROS, ...`, to avoid name clashes with existing code, these primitives could be defined as static `PrintFormat` class member and/or in a `_Print_` name space.

`Print` Public API

The public Print API is (in part) given by

```
Class Print
{
  unsigned int print(char);
  unsigned int print(char*);
  unsigned int print(char*, size);

  // for T ((signed|unsigned) (int|long|float|double)) | unsigned char
  unsigned int print(T item)                      {return print(item, DEFAULT) ; }
  unsigned int print(T item, unsigned char radix ) {return print(item, RADIX(radix)); }
  unsigned int print(T item, PrintFormat format);

  // corresponding println() methods will be defined by template functions
}
```

Summary/Recommendation/Discussion/Questions

My recommendations / opinion:

- Use a class named `PrintFormat` to specify format control instances that can be used by the `print()` methods. If name spaces are adopted, (see below), it may become `_Print_::Format`
- Use `operator&` for combining `PrintFormat` instances.
- Provide `operator~` option for format negation (if implementation does not pose a problem)
- Provide a `PrintFormat` constructor based on a comma separated list of `PrintFormat` instances.
- No support for methods like `.asSize(6).asHex(), .asFillZeros(), .asETC()`.
- Considering potential name clashes, storing names in a namespace is anyway a good idea. The name space can be activated by default by adding the '`use namespace _Print_;`' clause to the Print.h header file, making the name space transparent to the user. The question is, however, should this name space be activated by default.
- Any better candidate for _Print_ as the namespace name?
- With a grin: People who still long for the outdated `printf`, should buy themselves a RaspberryPie (where resources are not a concern).

`PrintSequence` Public API

I have a PrintSequence class implementation that we could package with the Print class. These code snippets are given as a teaser:

```
  int n = serial.getPrintSequence()
        <<" One   " <<1
        <<" Two   " <<2     <<NEWLINE   // *)
        <<" A     " <<3 <<4 <<endl      // *)
        <<" float " << 3.4  <<'\n'
//      <<" float " << HEX|FIELDSIZE(4) % 3.4            <<'\n'   // **)
//      <<" float " << PrintFormat(HEX|FIELDSIZE(4))% 3.4 <<'\n'   // **)
//      <<" float " << PrintFormat(HEX|FIELDSIZE(4), 3.4) <<'\n'   // **)
        <<  EndSequence();

//  *) apparently these are defined in some 'standard' include files.
// **) formatted concatenations are not yet defined or implemented.
//     I will do some research on how this is defined in the C++ standard iostream.
```

The following teaser shows how I intend to use this in a message based protocol.

```
char* messageHeader  = "{MJ_1:";
char* messageTrailer = "}\n");
PrintSequence myPrintSequence(messageHeader, messageTrailer);

n = myPrintSequence.start()
    << "TS=" <<timeStamp
    << ";T=" <<temperature
    << ";P=" <<pressure
    << ";h=" <<humidity
    << ";n=" <<noice
    << ";c=" <<co2
    << EndSequence();
```

Disclaimer,

I know that there are some incoherencies in the API's, these will be resolved.

Implementation schedule,

My proposed milestones are: (I will have a few free days in two weeks).

- Deliver, a (non Arduino) demo program that can be run from e.g. a Unix or Cygwin environment.
- Arduino `Print.h` (including, `PrintFormat, PrintSequence`) with class definitions, working with my actual `Print.cpp` version on github. Add a demo sketch.
- Fix a buglet in this `Print.cpp` version (`floats > 0x7fffff00` are currently truncated)
- Improve the performance and interplay between `Print.h` and `Print.cpp` (a bit of a bodge right now)

- Implement `printExponent()` in `Print.cpp`
- Make size and optimization studies to reduce footprint (assembly code inspection).
- Document API (through Javadoc style for autocompleting) (provided this works for templates as well).
- Format control for `printChars()`

Michael Jonker, 26 January 2015