

# Data Acquisition

Chris Brown

August 2010, March 2011

## Contents

<b>1 Data Acquisition Basics</b>	<b>2</b>
<b>2 The Hardware and Software Abstractions</b>	<b>2</b>
<b>3 Naming Channels</b>	<b>4</b>
<b>4 National Instruments USB-6009</b>	<b>5</b>
4.1 What They Tell You . . . . .	5
4.2 What They Don't Tell You . . . . .	6
<b>5 Case Study: 6009 and Analog Output</b>	<b>7</b>
<b>6 Readings</b>	<b>9</b>
<b>A Electricity: All You Need To Know</b>	<b>9</b>
<b>B Components</b>	<b>10</b>
B.1 Diodes . . . . .	10
B.2 Resistors . . . . .	10
B.3 Breadboard . . . . .	10
<b>C Number Bases</b>	<b>11</b>

# 1 Data Acquisition Basics

This is not a tutorial: we're not going into details. You're being given the "opportunity" (welcome to the world: that is code for "responsibility") to read existing technical documentation and understand it. Very luckily for us, the Matlab documentation is clear, has examples, is not just for experts, and is mirrored in exemplary on-line features and help. Your primary references are the PDF documentation "User's Guide" and (you may want to print this out) the "Quick Reference Guide", both found down near the bottom of <http://www.mathworks.com/access/helpdesk/help/toolbox/daq/daq.html>. (If you like video and html tutorials they are what you find higher up on this page.) If you're into hardware, our data acquisition (and voltage generating) device, National Instrument's USB-6009, is described in <http://www.ni.com/pdf/manuals/3713031.pdf>.

In the User's Guide, you should glance through the table of contents (pp. v —xvii) every now and then, and especially when you have a specific question in a particular subject context. Of course the Index is good if you have the right name of the subject at hand.

You should read the User's Guide, Chapter 1, and actually understand most of it. Pages 1-5 through 1-19 are full of basic concepts and are a good investment for your whole engineering career. You can skip from bottom third of 1-21 to 2nd half of 1-24: it's under-the-hood hardware issues that may be important to you sometime, though. Pages 1-24 through 1-45 are really basic principles and should be understood in detail, except we can ignore 1-31 through 1-33. In fact we shall be using exclusively the differential inputs for analog signals, which are generally preferred.

You may find Pages 2-19 through 2-25 to be quite useful: overviews, documentation examples and demos, the `daqhwinfo` function for getting information on your hardware, etc.

## 2 The Hardware and Software Abstractions

This is material from Chapter 3 of the Matlab DAQ User's Guide (the 744-page one). Details of the Analog Input, Output, and Digital I/O are in Chapters 4, 6, and 8, with even more detailed details and advanced stuff in Chapters 5 and 7.

Digital-to-Analog and Analog-to-Digital (D/A and A/D) converters are a lot alike, like cars or phones, and are accessed and manipulated in the Matlab DAQ toolbox (and LabView) in functional terms. But there are many DAQ products because their power and number of features vary (like cars and phones). Thus it behooves you to get the right product for your application (when you have a choice) on the basis of its specifications and what you can afford, and take note of how its hardware features are mirrored in the software. Our situation with one device is typical and representative and generalizable, so everything you learn about software access will serve you in good stead.

The USB-6009 we use is totally typical. However Matlab (or LabView, I bet) does not "understand it" directly. Even matlab needs the equivalent of a library of functions to deal with exterior hardware.

Each piece of computer hardware (monitors, disk drives, headphones, printers, DAQ hardware...) has a software *driver* that communicates with the computer, usually at the level of the operating system. Matlab has a uniform syntax that does not change with the device used, and so its commands have to communicate with these drivers. They do that via *adaptors*, usually one for each device manufacturer (plus the sound card). This layer is usually beneath your notice except for the one-liner you use to create an abstract version of your device: when you command `AO = analogoutput('nidaq', 'Dev3')`, for instance, you create an instance of the National Instruments adaptor to communicate with the 6009.

The LHS of this statement is any name you choose for the *analog input object* you have just constructed (don't panic, read on).

Both the RHS string arguments are “magic constants” dictated by our software. The 'nidaq' argument will not change for this class, it means we are using the National Instruments adaptor. The 'Dev3' string is a name for our 6009 device. *This string is subject to change after 2011.* Somehow the DAQ toolbox is remembering every installation of the software on campus, of which there have been three in three locations over the past year. The constant, in the other locations where the DAQ toolbox is still installed, is or has been 'Dev1' and 'Dev2'. For spring 2011 in Gavett 244 it's 'Dev3'.

So how do you know what's right? The software is pretty helpful (if you RTFM), and there's a powerful command `daqhwinfo()` that takes various arguments and gives you various info about hardware. If the DAQ toolbox does not recognize your device name, it gives you a red error message and points you to this command. If you type `>> daqhwinfo('nidaq');` at the Matlab command line you'll be told the name to use for the 6009 at your location.

The hardware has three basic functions.

1. Digital input and output: output turns on a voltage down a line (signaling a binary 1) or turns it off (signaling a binary 0). Input reads a voltage on a line and returns 1 if it is above some value and 0 if not.
2. Analog input: it reads a voltage in some range.
3. Analog output: it outputs a voltage in some range.

Matlab abstracts these functions into three different *objects*, which you use to control the device: `digitalio`, `analoginput`, `analogoutput`.

You'll notice the 6009 has many more than three connections. In fact you can do several of these I/O operations at once, thus sending or getting data to or from several sources “simultaneously” (not true, it cycles through them but we'll ignore that). Thus another level of truth and abstraction is the individual lines. Analog lines are called *channels*, maybe since they have more parameters you can set, and digital ones are indeed called *lines*. Thus each object has a set of lines or channels.

Each line or channel has parameters that determine its behavior. These are called *properties*. Some properties are common to all lines in the object, and can (must be) set all alike: these are the *common properties* of the object. Some properties are settable on a per-line or per-channel basis: these are *Channel (Line) Properties*.

That's where we stop: you'll read that the channel and line properties are both divided into *base* properties that are universal and *device-specific* properties that pertain to a particular device. We're only concerned here with base properties, so we ignore this distinction.

Important: Before you look at Analog Output, check out the situation with the USB-6009 (Section 5).

The properties we can control are things like the rate at which the device acquires analog data (for us, up to 48,000 samples / second), the voltage range we expect the data to be in (we have 8 ranges, from [-10 10] volts to [-1,1]), what starts the collection process (for us, whether the collection starts when we `start` the device or whether we start the device and want it to wait for an explicit `trigger` command to begin), and so forth. There are lists of properties, categorized by object and channel or line, in the DAQ reference manual and Quick Reference Guide.

Several commands give you information about what you can do, and some of them actually do things to the hardware: Say you have created a device by the command

```
ai = analoginput('nidaq', 'Dev3');
```

You can then use `inspect(ai)`, `get(ai)`, `set(ai)` for instance give you lots of facts (set tells what you can set, and get tells you what their current settings are) about the object `ai` you have constructed. To set a property you use syntax like `set(ai, 'PropName', <propval>)`, and to make sure the right thing happened you can use `ActualSetting = setverify(ai, 'PropName', <propval>)`. Other useful information commands are `daqhwinfo`, `inspect`, `propinfo`, ....

It seems that if you specify an out-of-bounds value for some property you either get an error message or the value is set silently to the one closest to the one you had in mind (there are only 8 discrete voltage ranges for analog input channels, so if you say you want range [-pi, pi], you'll get the next best thing (according to the adaptor.)

### 3 Naming Channels

DAQ abstractions are implemented as Matlab structures, and thus there are two basic syntaxes to access and update them. You should read Attaway Chapter 7 for the details, but basically a named structure is a bunch of named slots each of which contains a value. For example, if an analog input object is named `AI`, we can say `get(AI, 'SampleRate')` or `AI.SampleRate` and get the same result.

Not only that, you'll see that you can make up your own names for channels, for instance. At channel addition time there's an optional third parameter, a name.

```
addchannel(AI, 0, 'BlueLEDVolts')
```

After that you could say `AI.BlueLEDVolts.InputRange = [-5, 5]`, for instance.

You'll notice there is a bit of an issue naming channels. National Instrument channels are numbered from 0, so `addchannel(AI, 0:4)` creates five of them. However, Matlab addresses arrays from 1, so `AI.Channel(1)` here addresses Channel 0. So

`AI.Channel(1).Inputrange = [-5, 5]`) is how you'd address this channel as an array element.

The digitalio lines seem to be divided into two ports, P0 and P1. This is indeed the case, and you can choose to use the port distinction or ignore it. There are no port-specific properties, so maybe it's for parallel port operation...who knows? Anyway, you can do these sorts of things:

```
DIO = digitalio('nidaq', 'Dev3');
OutLines = addline(DIO, 0:3, 'out'); % Use Line No. Only
PortLine = addline(DIO, 1, 1, 'Port1Line1'); % Or Port/Line
sensor = addline(DIO, 2, 'in', 'OverHeat');
```

I'd recommend ignoring ports and just referring to lines 0-12. Thus to refer to line 0, also known as Port 0 line 0, given Statement 'Use Line No. Only', above, you'd use `OutLines(1)`. I think Port 1, line 1 is line 9.

Thus digitalio lines have matlab array and maybe symbolic names as above.

To output digital data, you may specify a 'bit vector' for which lines get 1 and which lines get 0: So you might see

```
putvalue(DIO, 12);
putvalue(DIO, logical([1 0 1 0]));
```

In the first command the digital 12 is interpreted in binary to get 1100. Note that the first high-order bit here refers to the first line in the line array (here `DIO.Line(1)`, which has `DIO.Line(1).HwLine = 0` (it's digital line 0 of the 6009)). Likewise the last (low order) bit in the vector refers to the last (highest-index-value) line in the Line array. Phew: see the User's Guide page 8-14.

This is Matlab, so you can create and address matrices of objects and channels with these commands. Not only that, but property names are case-insensitive and can be abbreviated, so 'SampleRate', 'SAMPleRate', or even 'SampleRa' are enough to identify the slot (abbreviations like the latter are a bad idea since some property like 'SampleRadix' might get added in the future).

## 4 National Instruments USB-6009

### 4.1 What They Tell You

This device (Fig. 1) has a lot going on inside it, including the complex process of dealing with the USB communications protocol, generating timing information, setting circuits for different voltages, etc.

Its main information page is this one at National Instruments:

<http://sine.ni.com/nips/cds/view/p/lang/en/nid/201987>. I've found most useful the little tab "Specifications", where you'll also find "Detailed Specifications" and "Data Sheet".

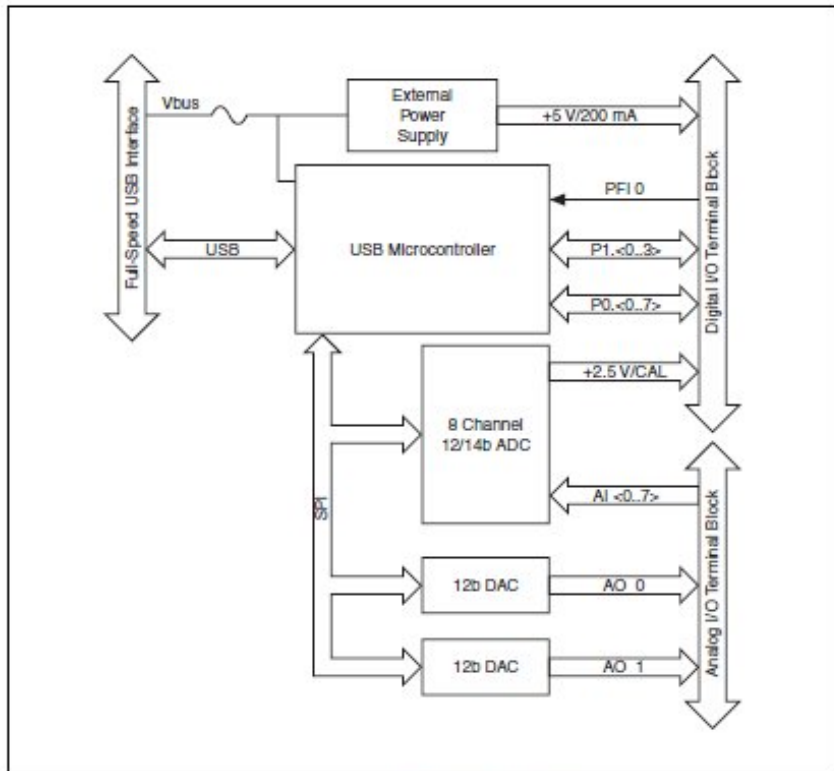


Figure 4. Device Block Diagram

Figure 1: The 6009 and its Schematic.

All this is useful in some off-line circumstances, but from a practical point of view, the DAQ toolbox, via commands like `get`, `set`, `setverify`, `daqhwinfo`, `inspect`, `propinfo`... can tell you all the attributes you can access, change, or need to know.

A possibly useful page on the labeling of the lines and channels is <http://techt teach.no/tek dok/usb6009/index.htm>.

## 4.2 What They Don't Tell You

; The 6009 has internal state that sometimes gets set mysteriously.

IF you get strange errors involving timeouts, lack of timeouts, operations not finished, or samples missed, if `get(ai)` (for instance) shows the object in the 'Running' state when it shouldn't be, if you're told to kill off other jobs or set `SampleRate` lower, or you get other mysterious whining,

AND your code seems OK,

THEN `delete(daqfind); clear;`, turn off connections into the 6009 that may have voltages on them,

DISCONNECT the 6009 from its USB cable and reconnect. That should fix things. If not, call for help!

## 5 Case Study: 6009 and Analog Output

This true story illustrates a little sleuthing and real life use of DAQ commands.

In *general*, (and in the User's Guide) analog output works like this: First, you can set an output voltage on a channel with a `putsample()` command. Or second, you can queue up (in the adaptor-driver 'engine') a vector (say *MyData*) of analog voltages you want emitted over your channel by using a `putdata(ao, MyData)` command. You set the rate for these outputs using `set(ao, 'SampleRate', ...)`. After starting the ao object, you issue a `trigger(ao)` and the values are sent down the line at the specified rate, in complete analogy to the way an analog input object works.

BUT. The 6009 can't do the `putdata()` thing. How did I discover this? Turns out if I'd just blundered ahead and stubbed my toe I would have gotten a very helpful error message telling me the above sad fact and advising me what I should do instead. Gotta Love Matlab's User Interface, I must say.

Being me, I wanted to find out just what the timing ranges were before I jumped in. My first hint of a mystery was NOT in the spec sheet on the main 6009 page. That says you can go up to 150 samples/sec on an AO line, but no lower range. The "detailed specifications" PDF has electronic schematics and the words "software timed" opposite the same 150 S/s line in the specs sheet. This doesn't look good, but it could mean anything, like maybe the driver does it. The schematic for the analog output circuit looks suspiciously simple, though. My experts in ME who've used the 6009 for years tell me: "no idea," and "I always put analog input in a while loop," which again doesn't sound good but isn't conclusive.

So in the lab I try some stuff: I made a digitalio object and:  
`setverify(dio, 'TimerPeriod', 0.001);`  
seemed to work, but that's faster than 150/second. What's up?  
`propinfo(dio, 'TimerPeriod');`  
gives me a range from [0.001 – 2 billion]. Oops...this is interesting and I didn't know these numbers, but I really meant ANALOG output, not digital: "never time to do it right, always time to do it over."

I make analogoutput object ao, try: `inspect(ao)`,  
which shows 150 samples each channel... no news and no help.  
`setverify(ao, 'SampleRate', 100)`  
works fine, and  
`propinfo(ao, 'SampleRate')`  
returns [.6 150], which looks believable.

So I'm ready: I make a script like this:

```
ao = analogoutput('nidaq', 'Dev3');
addchannel(ao, 0, 'aout');
ActualOutRate = setverify(ao, 'SampleRate') = 100 % trust but verify
GotOutRate = get(ao, 'SampleRate') % another check
set(ao, 'TriggerType', 'Manual');
```

```
VoltsOut = linspace(-1,5,100);  
putdata(ao, VoltsOut);  
get(ao, 'SamplesOutput') % 100 if queued or sent? not sure  
start(ao);  
trigger(ao);  
wait(ao, 3); % should only take a second  
delete(daqfind);
```

**BOOM. Error at the putdata command to the effect: “Your hardware does not support timed output. You must use software equivalent. Most systems support pauses down to 0.01 second. Sorry for any inconvenience and have a nice day.”**

There it is. So I tried the following script several times, and you might too.

```
tic  
pause(0.01)  
toc
```

Don't know about you, but I got a bunch of .0092 and .0096, one .019... it's like we're right at the quantization limit for Matlab's pause here, but more research needs to be done maybe. One thing we'll do is the final LED lab.

**Conclusion: we can forget the interesting version of analog output for the 2009. We can only use putsample, pause, and prayer.**



## 6 Readings

[1] *Transducer Interfacing Handbook A Guide to Analog Signal Conditioning*, edited by Daniel H. Sheingold; Analog Devices Inc., Norwood, MA, 1980.

[2] Bentley, John P., *Principles of Measurement Systems, Second Edition*; Longman Scientific and Technical, Harlow, Essex, UK, 1988.

[3] Bevington, Philip R., *Data Reduction and Error Analysis for the Physical Sciences*; McGraw-Hill, New York, NY, 1969.

[4] Carr, Joseph J., *Sensors*; Prompt Publications, Indianapolis, IN, 1997.

[5] *The Measurement, Instrumentation, and Sensors Handbook*, edited by John G. Webster; CRC Press, Boca Raton, FL, 1999.

## A Electricity: All You Need To Know

What follows is no substitute for your high-school physics. Electricity is a *current* of electrons thought of (for some strange reason) as flowing from a *positive* source into a sink called *ground*. It is measured in *amperes*, proportional to electrons per second through the wire. The more electrons per second, the more current. The force propelling the electrons is *electromotive force*, or a *voltage*, or a *potential difference* (measured in *volts*, as from a battery or other source).

We'll mainly deal with *direct current*, which flows in one direction. *alternating current*, found in house wiring or the input and output of *transformers*, flows alternately in both directions, in the USA usually at 60 Hz. Later we'll use a step-down transformer that produces 6 volts of alternating current from the 110 volt AC from the wall plug.

The current in a wire can be lowered by a *resistor*, which converts some of the electrons to heat. Resistance is measured in *ohms*, and typical resistors have values from a few ohms to millions. Current, voltage, and resistance are governed by *Ohm's Law*: If  $I$  is current,  $R$  is resistance, and  $V$  the voltage (potential difference) across the resistance, then  $I = V/R$ .

That's it. The 6009 has a +5 volt voltage source and several ground terminals. The digital outputs come with a ground connection and put out something like 6 volts relative to that ground when they are turned on. The digital inputs register 0 if the input voltage is low, and 1 if it is high (again, above about 6 volts). The *single ended* analog inputs measure the difference between one input and the associated ground terminal. The *differential* analog inputs measure the potential difference between their two inputs, and are generally more accurate and to be preferred. The analog outputs put the requested voltage on their positive terminal, relative to the associated ground terminal. All the 6009's grounds are connected but there are several for ease of wiring.

## B Components

### B.1 Diodes

Diodes pass current one way and don't want to pass it the other. They are asymmetrical, and must be wired with that in mind. Our diodes are light-emitting diodes: if their positive end is connected to a positive voltage source and their negative end is connected to ground, they will emit light as long as the positive voltage is in some range (about 1 volt to 5 volts). If they get too much voltage in that direction, they could burn out (depending on the current). If they are wired in the opposite direction, they'll either stop the current or blow out. Morals: wire them in the right direction and control their input voltage.

They tend to have two clues to their polarity. Their long lead is for the positive input, the shorter one is to go to ground. Also they tend to have a flat part on their bulb, which is on the ground side.

To limit the voltage across them, we put them in *series* with a resistor. That is, a resistor is between them and the voltage source and the ground. It can be before or after them in the circuit, that's not important. We'll use any old resistor in the 500-700 ohm range for an input of 10 volts and a current (supplied by the 6009) of 200 milliamperes.

### B.2 Resistors

Resistors are symmetrical devices, and can be facing either way in a circuit. Their values in ohms are encoded by the colored bands on their bodies. Google 'resistor color code chart' or go to [http://www.elexp.com/t\\_resist.htm](http://www.elexp.com/t_resist.htm) for instance.

For our work we'll give you the resistors you need.

### B.3 Breadboard

This useful item lets you put together temporary circuits and change them easily (no soldering). Fig. 2 shows that in this case there are semi-rows of five holes and vertical columns of 30 holes, in which each hole is electrically connected to the others.

You can see that the rows are numbered from about 0 to 30 and that there are two columns on the left and right of each row. The internal columns are labeled in two groups, A-E on one side and F-J on the other.

In the figure, the far left and far right two columns (of about 30 holes each) are connected: the blue and red lines on the breadboard are strictly for your convenience, but they are sometimes used as a common positive voltage source (connected to any hole in the red column) and ground (connected to any hole in the blue). We won't be doing that uniformly, but we could.

In each of the 30 rows, the holes A-E and F-J are also electrically connected. None of these internal 5-hole sub-rows is connected to any other hole: all 5 holes in a sub-row y are

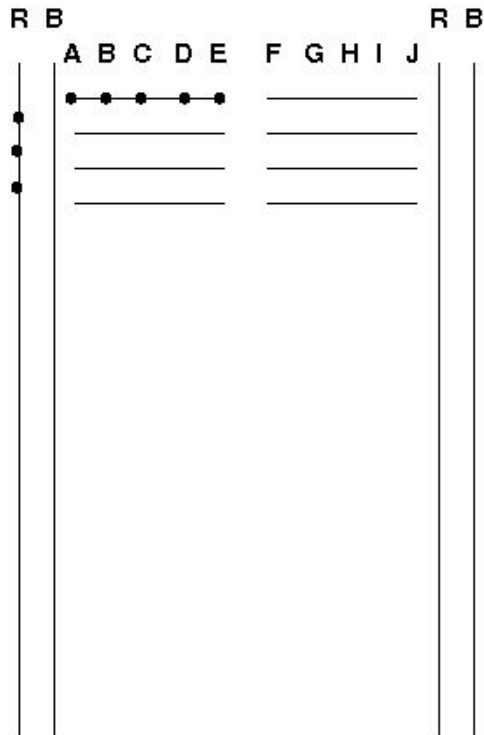


Figure 2: Typical solderless breadboard. Lines indicate semi-rows and columns of holes, for each of which the holes are electrically connected.

connected to each other but isolated from all other holes – in particular the A-E holes on row X are not connected to the F-J holes on row X or any other row.

To connect a component (LED, resistor, or wire) to a hole just jam the uninsulated end of the lead into the hole.

## C Number Bases

In a number-base system (for instance, the base-10 or decimal system in worldwide use) a non-negative integer (like 327) is written as a string of *digits*  $d_i$  (here  $d_0 = 7, d_1 = 2, d_2 = 3$ . That string is here a base-10 (decimal) representation of the integer that is the three hundred and twenty-seventh successor of zero, or the sum of  $325 + 2$ , or generally the integer we agree we mean by 'three hundred and twenty-seven'.

To define what the string of digits means, the rule is:

$$I = \sum_{i=0}^n d_i b^i$$

where  $b$  is the base of the number system. Thus  $327 = 7 + 2 \cdot 10 + 3 \cdot 100$ , and we say 2 is in the “ten’s place” and 3 is in the “hundred’s place”, etc.

How many different digits do we need? As many as the base: in base 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. After that, we go to two digits: 10, 11, 12... And the 'largest' digit is one less than the base.

That's it. Integers in 'trinary' look like 210212, and in 'octal' they look like 67045300215, for example, and their values are computed as above. It's not easy for us to get their English equivalents by inspection since we don't usually have powers of 3 and 8 memorized.

*Binary, octal, and hexadecimal (base 16)* numbers are important in computers since binary numbers are the basic representation of computer data and both octal and hexadecimal act like shorthand for binary.

The first eight binary numbers are

```
421 <-- "binary places"
```

```
bin  dec
000  0
001  1
010
011
100
101  5
110
111  7
```

Often in computer representations (as in naming lines in the USB-6009 DA/AD converter), a *bit string* like 001001 represents a set: in this case we have a domain of six items and the string is 1 in the first and fourth (counting from right) positions, so it denotes the set consisting of the first and fourth items from the domain. Sometimes the digits in the string are called 'flags', with the understanding that 1 means the flag is true and 0 if not. Thus for instance the Unix permission codes for a file is (about) a string of 9 bits:

```
owner          group          others
read write execute  r w x      r w x
1    1    1    1 1 0    1 0 0
```

The above string means the file owner has total control, members of his group can read and write the file, but others can only read it.

Clearly, a 32-bit integer is a string of 32 bits, this time to be interpreted (by hardware) as binary digits, not flags.

Octal and Hex are shorthand for binary since their bases are powers of two. Each octal digit represents three binary bits. 0 for 000, 1 for 001, 5 for 101, and 7 for 111 for instance. Likewise there are 16 Hex digits, universally written 0-9, A,B,C,D,E,F: 3 is binary 0011, 9 is 1001, A is 1010, F is 1111. Thus we can write 32 bits as 8 Hex digits like DEADFADE or FEE169F0.

Famous CS riddle: Prove Halloween = Christmas.

Mind-stretching department: Consider negative, imaginary, and complex number bases!  
The same old formula above applies.