# Binary Number Representations and You

As you begin your ECE 194D labs, you will encounter strange new data types in your Simulink models. Where did these uint8's and int32's come from, and why can't you just use doubles for everything anymore? For that matter, what is a double? And why should you care about these persnickety low level distinctions anyway? Numbers are numbers, right?

Well, young student, prepare to have your interest piqued. The software we're using for the ECE 194D labs has the potential to be *incredibly* frustrating if you do not understand how to use these different data types. Why? When you design a Simulink model to control your robot, that block diagram gets converted into a computer programming language called C before being compiled and downloaded into your robot. In the past, when you've used Matlab or Simulink, you've probably never had to worry about variable types. Everything was handled implicitly in the background, transparently.

However, C handles nothing transparently, nor in the background. Since your Simulink block diagram is getting converted into C, you need to do some hand holding to make everything work. This document will help you understand how to do this.

## *A Brief Primer on Binary Numbers*

We usually represent numbers by writing out their digits. For various historical and anatomical reasons, most humans prefer base ten (decimal) numbers. This means we use ten distinct symbols (or digits), strung together with additional helper symbols like decimal points and minus signs, to represent numbers. For example: 3, 192.4, -10.2 are all base 10 numbers.

Computers, however, prefer base 2 (binary) numbers. Here, the numbers are represented as a sequence of 1's and 0's. Lets say you have room to store 3 of these **bi**nary digi**ts** (bits). There are $2^3 = 8$ different combinations of these three bits, listed below.

| Bit 2 | Bit 1 | Bit 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

*Table 1: Every possible combination of 3 bits*

Lets say we want to use these three bits to represent the integers from 0 to 7. We need to come up with a one-to-one mapping from the 8 different combinations of 3 bits to the 8 integer values we're interested in. There are a huge number of possible ways[1] to choose this map, but in practice only a few are used because they have properties that make it straightforward to do binary arithmetic.

1 For n bits, there are $(2^n)!$ distinct mappings. For n=3, this is 40,320. For n=32, you don't want to know.

## *Unsigned and Signed Types*

In the previous section, we talked about mapping a sequence of n bits to a set of $2^n$ integer values. I mentioned that only a few maps are commonly used. We'll talk about two of these mappings in this section.

**Unsigned Binary Numbers**

Unsigned binary numbers are used to represent positive (or zero) valued integers. We already know that if you have n bits, you can put the n bits in one of $2^n$ configurations. This allows us to represent every integer from 0 up to ( $2^n-1$ ) as a unique combination of n bits (in a particular order with special properties). Such a representation is referred to using the shorthand uint*n*, where the u stands for "unsigned[2]", the int stands for "integer" , and *n* is the number of bits. Therefore, for 3 bits, uint3 is the shorthand name[3].

| Bit 2 | Bit 1 | Bit 0 | Decimal Value (uint3) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

*Table 2: uint3 mapping*

Question: What are the lowest and highest integer values that can be represented by variables of type uint8, uint16, and uint32?

Question: Assume the world population is 7 billion.

a) What is the smallest value of n such that you can assign each person a unique n-bit binary number?

b) People find binary numbers dehumanizing. The UN wants you to come up with a way to convert these binary numbers into familiar decimal numbers. You decide to use the unsigned integer representation, since its the only one you know. Given your answer in part a, is uint32 an appropriate choice? Why?

2 "Unsigned" because there is no sign information stored. Everything is assumed to be positive (or zero).

3 uint3 does not exist in Matlab, but uint8, uint16, uint32, and sometimes uint64 do.

**Signed Binary Numbers**

The last section showed us how to represent positive integers in binary, but what about integer values that could be negative or positive? We normally tack a minus sign onto a number to denote negativity, but computers have only ones and zeros, not ones and zeros and minus signs. How is negativity represented with binary numbers?

The short answer is that we just map some of the binary combinations to negative numbers and some to positive numbers, instead of all to positive numbers as before. The details of this mapping (and why the particular one we use has really useful properties) are quite tedious, but if you're curious there's a good summary here: http://en.wikipedia.org/wiki/Two's_complement

A signed n-bit binary number is often referred to using the shorthand int*n*, along the same lines as uint*n* for unsigned numbers. An int*n* can represent integer values from $-2^{(n-1)}$ to $2^{(n-1)}-1$ . So, what does the int3 mapping look like?

| Bit 2 | Bit 1 | Bit 0 | Decimal Value (int3) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | -4 |
| 1 | 0 | 1 | -3 |
| 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | -1 |

*Table 3: int3 mapping*

Notice that the first four entries are identical to the uint3 mapping. As long as your decimal number is small enough ( <= $2^{(n-1)}-1$ ), the binary representation is identical whether you use uint*n* or int*n* as your data type. Above this limit, the two mappings represent different numbers, which can cause problems if you use mix them up in your program. If you see very large negative (or positive) numbers show up in your data, this has probably happened.

Question: What are the lowest and highest integer values that can be represented by a int8, int16, and int32?

Question:

a) Your friend has the 3-bit binary number 101, but isn't sure if the proper data type is a uint3 or an int3. You decide to assume it should be interpreted as a **uint3**. What is the corresponding decimal value?

b) If the same binary number is interpreted as an **int3**, what is the corresponding decimal value?

**Overflow, Saturation, and Wraparound**

As you have seen, a (signed or unsigned) n-bit binary number can only represent integers within a certain range. What happens when you try to represent a number that is too large? These issues frequently come up when doing arithmetic with integer data types in Matlab and Simulink.

Imagine you have the numbers 54 and 106, and you choose to store these values as int8's. There's no problem with this, since they're both within the allowable range of -128 to 127. However, if you compute the sum of these variables in Matlab you'll expect 160, but get 127:

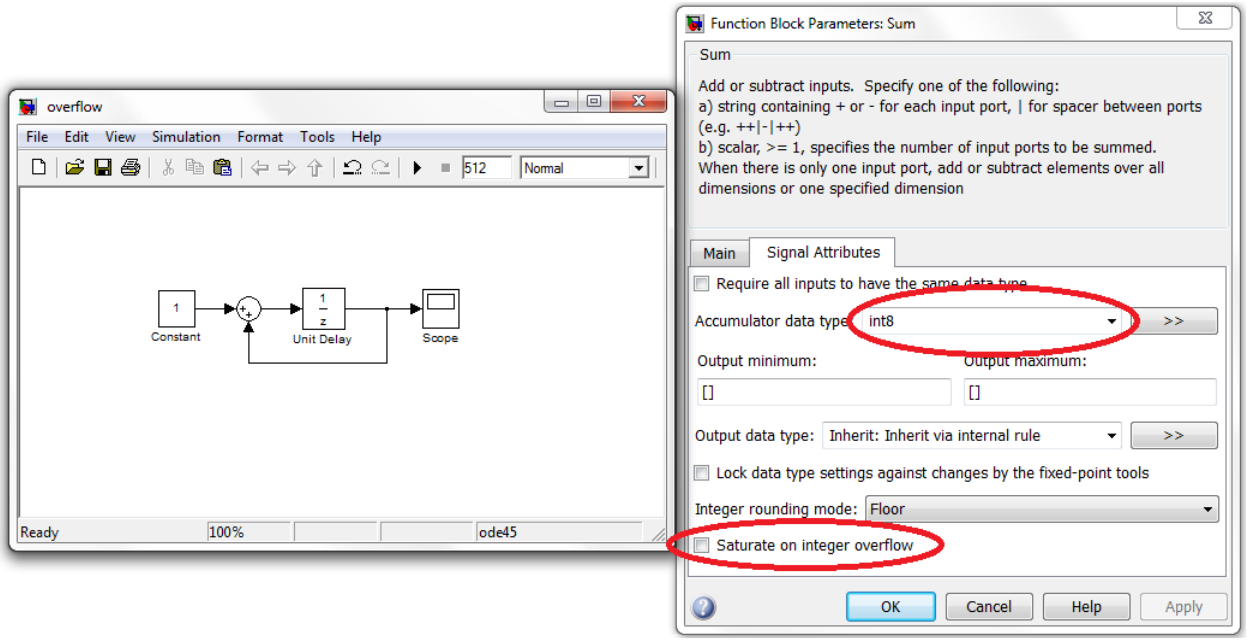
```
>> int8(54) + int8(106)
ans =
  127
```

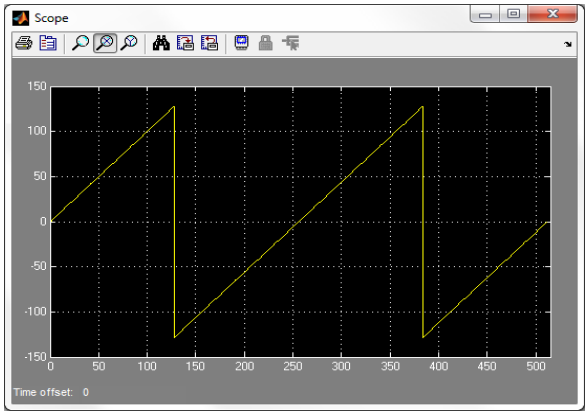
The sum produced an *overflow*, which means the resulting number is too large to store in an int8. Matlab responded to this error by *saturating* the result at the closest allowable value, which is 127. Among programming languages, this is unusual. The other way a computer can handle an overflow is wraparound. Imagine connecting the largest allowable value back to the smallest. When a value exceeds the largest allowable value, it is wrapped back to the smallest (most negative). There are situations where this is desirable, but usually it causes problems.

Imagine that control software on a $370 million dollar rocket is tracking its own velocity using a signed integer. Being a rocket, the velocity will become some increasingly huge positive number. However, imagine that at some point that huge positive number becomes *too* huge, and an overflow happens. The computer responds by wrapping that number around to a huge *negative* number, which means the rocket thinks it is flying the wrong direction very fast. The control system attempts to compensate for this enormous error (i.e., turn around), ultimately causing the rocket to break up from aerodynamic stresses. The software you create in ECE 194d is susceptible to the same types of problems (if not the consequences), so pay attention.

*Figure 1: Ariane 5 rocket explodes in 1996 due to overflow and wraparound in 16 bit signed integer*

Question: You create the following discrete time integrator system in Simulink, and get the following simulation result.




a) This is probably not what you were expecting from an integrator. Why did the system produce this output?

b) What would happen if you checked the "Saturate on integer overflow" box and ran the simulation again? What will the output be after 1000 time steps? Why?

## *Floating Point Numbers*

Normally, Matlab and Simulink prefer to make variables "doubles". This is a very flexible data type. It can store positive numbers, negative numbers, and numbers of extremely large and small magnitudes. Most importantly, doubles can represent real numbers, not just integers (although the representation is subject to small rounding errors).

Doubles are an example of "floating point" numbers, which like signed and unsigned integers are stored as a bunch of bits. However, the various combinations of bits are mapped to locations on the real line instead of to integers.

Here's a very brief explanation for how they work (although you don't need to understand this to use them). Imagine you have n bits for the whole floating point number. Divide these bits into two groups, which you think of as representing two signed numbers: X and Y. To compute the decimal value of the floating point number, you calculate $X * 2^Y$ . This is conceptually similar to scientific notation. It allows very large numbers, as well as numbers of very small magnitudes, to be represented by choosing an appropriate exponent (Y), and significand (X).

Why is the data type called a double? There is another floating point type called a single, which takes 32 bits to store. A double uses the same general format, but allows for a larger range because it is allowed to take up 64 bits, which is "double" the size of a single. The minimum and maximum values for these data types is shown below.

| Type | Minimum Value | Smallest Value | Maximum Value |
|---|---|---|---|
| single | -3.40E+038 | 1.18E-038 | 3.40E+038 |
| double | -1.80E+308 | 2.23E-308 | 1.80E+308 |

*Table 4: Floating point data type ranges*

Why not use doubles for everything in your Simulink model? The processor used in the NXT brick does not have hardware support for floating point operations, so every floating point operation you tell it to do gets translated into a large number of integer operations, which takes time for the CPU to carry out. There's only a finite amount of time for the CPU to do everything between samples, so don't go overboard with floating point operations. If you do need floating point numbers, consider using a single to save time and memory. (If you can get away with it. Consult Table 4 and Figure 1.)

**Summary**

| Type | Minimum | Maximum | Bits | Integer or Real? | Note |
|---|---|---|---|---|---|
| uint8 | 0 | 255 | 8 | Integer | Watch out for overflow! |
| uint16 | 0 | 65,535 | 16 | Integer | |
| uint32 | 0 | 4,294,967,295 | 32 | Integer | |
| int8 | -128 | 127 | 8 | Integer | |
| int16 | -32,768 | 32,767 | 16 | Integer | |
| int32 | -2,147,483,648 | 2,147,483,647 | 32 | Integer | |
| single | -3.40E+038 | 3.40E+038 | 32 | Real | No NXT hardware support. More CPU intensive, subject to rounding errors. |
| double | -1.80E+308 | 1.80E+308 | 64 | Real | |

## *Simulink Tips: Data Type Conversion Blocks*

Use Data Type Conversion blocks any time you are connecting blocks which generate or accept different data types. For example, the revolution sensors output an int32, but the servo motors take in an int8. Here's a Simulink model:

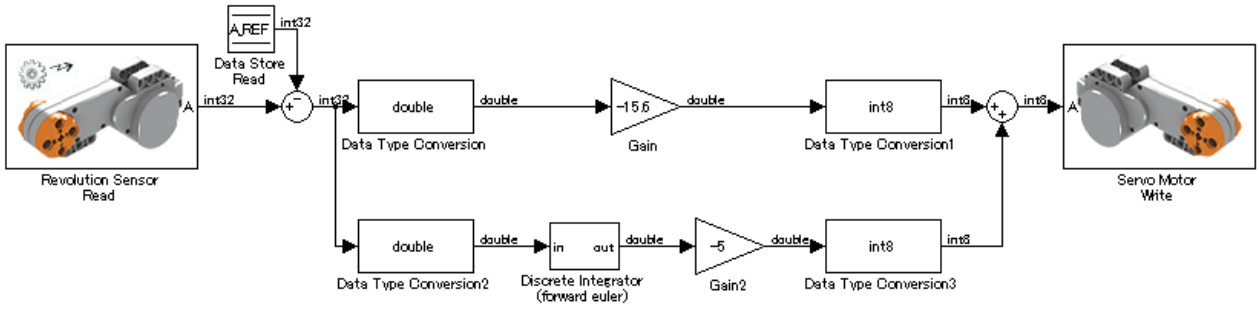



*Figure 2: Use Data Type Conversion Blocks liberally to prevent code generation errors*

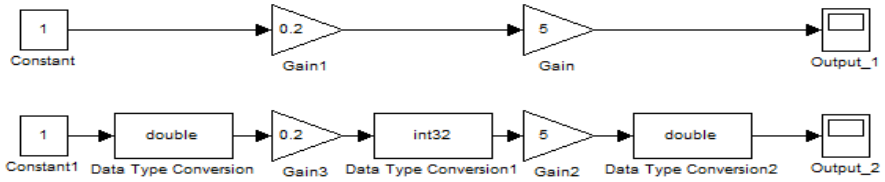
However, when you convert floating point variables (doubles) back to integers, there will necessarily be a rounding error. While lots of data conversion blocks will help Matlab generate your code, the code might do the wrong thing if these rounding errors are significant enough. The next few questions will help you understand the benefits and drawbacks of using Data Type Conversion blocks.

Question: When you convert a double to an integer, there is a rounding error. What parts of the block diagram in Figure 2 are subject to these rounding errors? (circle them)

Is there a way to change the block diagram to reduce this problem?

Question: Given that converting from one data type to another takes CPU time, how else might you minimize the number of conversion blocks in Figure 2?

Question: Consider the following Simulink block diagram.




a) What will the (constant) output be at the scopes Output_1 and Output_2? Write the intermediate values at the output of each block.

b) Why are the two outputs different?

Question: The Segway robot you'll use in lab (right) has wheels with a circumference of about 20cm. The wheels are attached directly to an encoder that measures their angular position in increments of one degree, so that there are 360 degrees per revolution of the wheel. The angular position is cumulative, so that if you spin the wheel around twice, the encoder will output 360*2=720 degrees.

The robot is programmed to drive forward in a straight line forever. If the wheel's angular position is stored in a variable, how far can the robot drive forward without experiencing an overflow if:

a) the variable is a uint8?

b) the variable is an int32?

c) the variable is a uint64?

d) Represent these distances, as appropriate, on these depictions of a loaf of bread, the United States, and the distance between the sun and the next closest star, Proxima Centauri.