**Floating Point versus Fixed Point Numbers**

Most of us are familiar with the integer data types used by the Arduboy environment – of various lengths, both signed and unsigned. We take care to choose appropriate data types that consider both the range of our data as well as the memory required to store them. Operations involving integer values are incredibly efficient and for many sketches are all that are required.

Sometimes however, integer values are simply not flexible enough. Imagine trying to calculate the movement of an accelerating rocket or calculate the trajectory of a missile in an Arduboy version of Choplifter. (Now there’s a good idea for an Arduboy project!) To make these calculations, you need numbers with both an integer and fractional component. The typical way to represent such numbers on a computer is through floating points.

C++ includes two floating point data types of different precision – the float and the double. On most platforms the float is 4 bytes and the double is 8 bytes, however on the Arduino (and thus the Arduboy) they are actually same precision (4 bytes) and are therefore interchangeable. These data types can store numbers in the range of -3.4028235 x 1038 to +3. 4028235 x 1038 (for comparison there is estimated to be 7.5 x 1018 grains of sand in the world) but at a precision of only 6 or 7 decimal digits.

For the Arduboy, in contrast to integer operations, floating point calculations are internally quite complicated and thus incredibly processor intensive. Including numerous calculations to reposition the player and enemies within the main loop of a program might prevent the game from performing well or worse yet even being playable.

However, there is an alternative to **floating point** arithmetic. This alternative is **fixed point** arithmetic. Fixed point numbers behave more similarly to the integer values you are familiar with. As such, the processing power required to manipulate fixed point numbers is more similar to integer calculations than to floating point calculations. Using fixed point numbers, that version of Choplifter might be possible after all!

There is however a tradeoff for the benefit that fixed point data types bring. The tradeoff is that you must explicitly nominate the size of both the integer and fractional portion of the variable – much in the same way you nominate the available range when choosing integer data types. In addition to the obvious data range restrictions of the integer component, the resolution (or 'granularity') of the fractional component must also be chosen.

For example, an unsigned 8 bit fixed point type with a 4 bit integer part and a 4 bit fractional part would permit an integer range of 0 to 15 and a fractional range of 0/16 to 15/16. This equates to a range of 0 to 15.9375 (15/16 is equivalent to 0.9375).

Note that the fractional part can only go up in increments of 1/16 (i.e. 0.0625), hence that particular format cannot accurately represent numbers at a lower resolution, for example the value 0.05 could not be accurately represented, it would have to be rounded to 0.0625. However, by using larger types or by increasing the number of bits afforded to the fractional part at the expense of the integer part you can balance the format to suit your needs.

For example, an unsigned 16 bit type with 8 bits of integer part and 8 bits of fractional part could represent all the positions on the Arduboy's screen, whilst providing a reasonable range of fractional steps in between the precise pixel locations. This would mean having much smoother movement than using regular integers, but in a way that's cheaper than floating points in terms of both code use and processing power.

**Fixed Points in Action**

In the examples below, I am using an early alpha release of @Pharap’s FixedPoints library.

@Pharap’s library defines a number of signed and unsigned data types which are listed at the end of this article. It also includes overrides for the common arithmetic and comparison operators (+, -, / , \*, <, >, etc) as well as some handy functions like ceilFixed(), floorFixed() and roundFixed().

Please note that an extra 'level of precision' argument is being used in the print code here. Without supplying this extra argument, the printed results suffer from rounding errors made by the Serial.print and Serial.println functions to simplify the code for printing floats.

Firstly, some simple assignments and calculations.

Code:

SQ7x8 myVar = 7.0;

myVar = myVar + 3;

Serial.print("static\_cast<float>(myVar) : ");

Serial.println(static\_cast<float>(myVar), 4);

Output:

static\_cast<float>(myVar) : 10.0000

Simple. Now let’s introduce a decimal place.

Code:

SQ7x8 myVar = 3.7;

Serial.print("myVar.getInteger() : ");

Serial.println(myVar.getInteger());

Serial.print("static\_cast<float>(myVar) : ");

Serial.println(static\_cast<float>(myVar), 8);

Output:

myVar.getInteger() : 3

static\_cast<float>(myVar) : 3.69921875

But hang on, we set the variable to 3.7 and it printed out 3.69921875. Why so? As mentioned earlier, both **floating point** and **fixed point** numbers introduce a degree of inaccuracy when storing numbers. With Fixed Point numbers, the precision (and by extension the accuracy) is specified by the programmer. In the sample code above, the variable chosen was a two-byte incarnation and it can store a number roughly between -128 and +128. The integer component is stored in 7 bits which allows a range of 0 to 127 and the fractional part is stored in 8 bits and has a range between 0 / 256 to 255 / 256. The largest fractional portion (255 / 256) converted to decimal is 0.9961 hence the data range cannot quite store +128, but it can store up to 127.9961.

If you could view the bits in the fractional part of the variable (which you can through myVar.getFraction()), you would see that it is stored as 179. As it turns out 179 / 256 is equal to 0.69921875.

The inaccuracy introduced by a relatively small (two-byte) **fixed point** is irrelevant if all you are trying to do is simulate some realistic movements of characters in a game. If you have a need to calculate more accurate values, you can simply increase the size of the **fixed point** variable to either an SQ15x16 or an SQ31x32 (though be aware that some operations do not work on SQ31x32 due to presently unavoidable restrictions). The code below illustrates the accuracy of each data type.

Code:

SQ7x8 myVar7x8 = 4;

for (uint8\_t i = 0; i <= 10; ++i) {

Serial.print(static\_cast<float>(myVar7x8), 8);

Serial.print(' ');

myVar7x8 = myVar7x8 + 0.1;

}

Serial.println();

SQ15x16 myVar15x16 = 4;

for (uint8\_t i = 0; i <= 10; ++i) {

Serial.print(static\_cast<float>(myVar15x16), 8);

Serial.print(' ');

myVar15x16 = myVar15x16 + 0.1;

}

Serial.println();

SQ31x32 myVar31x32 = 4;

for (uint8\_t i = 0; i <= 10; ++i) {

Serial.print(static\_cast<float>(myVar31x32), 8);

Serial.print(' ');

myVar31x32 = myVar31x32 + 0.1;

}

Serial.println();

Output:

4.00000000 4.09765625 4.19531250 4.29296875 4.39062500 4.48828125 4.58593750 4.68359375 4.78125000 4.87890625 4.97656250

4.00000000 4.09999084 4.19998168 4.29997253 4.39996337 4.49995422 4.59994506 4.69993591 4.79992675 4.89991760 4.99990844

4.00000000 4.09999990 4.19999980 4.30000019 4.40000009 4.50000000 4.59999990 4.69999980 4.80000019 4.90000009 5.00000000

One thing to note with the output is the compounding effect that adding two inaccurate numbers results in. This can be seen easily if we repeat the same exercise on the smallest, one-byte data type as shown below. The fractional part of an SQ3x4 number is stored in 1/16th increments or 0.0625 in decimal. When we add the first increment of 0.1 we immediately introduce an inaccuracy as the number gets rounded down to the nearest 1/16th of a number. As we repeat the addition, the error increases. The following code shows that.

Code:

SQ3x4 myVar3x4 = 4;

for (uint8\_t i = 0; i <= 10; ++i) {

Serial.print(static\_cast<float>(myVar3x4), 4);

Serial.print(" ");

myVar3x4 = myVar3x4 + 0.1;

}

Output:

4.0000 4.0625 4.1250 4.1875 4.2500 4.3125 4.3750 4.4375 4.5000 4.5625 4.6250

Choosing larger **fixed point** data types increases the range of the number and accuracy of the fractional component at the expense of memory.

**Why would I bother with Fixed Points at all?**

Imagine you are building a sideways-scrolling game where aliens (or cars, planes or other moving objects) are moving towards you at various speeds.

With each pass through the program’s loop you could decrement the x coordinate of each alien but this would result in all aliens moving at the same speed. You could try to move aliens every second or third time through the loop by keeping some sort of counter but this can be error prone and may result in the aliens moving slowly. To really allow flexibility in the aliens' movement, you could store the x and y in fixed point variables and increment these by a decimal value allowing the aliens to move at similar but slightly different speeds.

Consider the Alien class below. Each alien is described with an x and y coordinate and takes care of its own movement.

Code:

class Alien {

public:

Alien(SQ7x8 x, SQ7x8 y, SQ7x8 incX, SQ7x8 incY, const uint8\_t \*bitmap) {

\_x = x;

\_y = y;

\_incX = incX;

\_incY = incY;

\_bitmap = bitmap;

}

SQ7x8 getX() const { return \_x; }

SQ7x8 getY() const { return \_y; }

void setX(const SQ7x8 value) { \_x = value; }

void setY(const SQ7x8 value) { \_y = value; }

inline void Alien::move(uint8\_t frame) {

\_x = \_x - \_incX;

\_y = \_y - \_incY;

Sprites::drawOverwrite(\_x.getInteger(), \_y.getInteger(), \_bitmap, frame);

}

private:

SQ7x8 \_x;

SQ7x8 \_y;

SQ7x8 \_incX;

SQ7x8 \_incY;

const uint8\_t \*\_bitmap;

};

And in our main .ino file.

Code:

Alien alien1 = {127, 16, (SQ7x8)0.9, (SQ7x8)1.34, alienBitmap };

Alien alien2 = {127, 32, (SQ7x8)1.7, (SQ7x8)1.76, alienBitmap };

Alien alien3 = {127, 48, (SQ7x8)1.5, (SQ7x8)2.13, alienBitmap };

void loop() {

alien1.move(frame);

alien2.move(frame);

alien3.move(frame);

};

With **fixed point d**ata types, we can have our aliens move at slightly different speeds without the overhead of using floating points.

**Common Type Aliases**

@Pharap’s library includes the following data types. However the library is implemented with templates and it is a relatively simple matter to declare your own type aliases.

The predeclared aliases (which can be used as an example of how to declare such an alias) can be found in src\FixedPointsCommon\SFixedCommon.h and src\FixedPointsCommon\UfixedCommon.h.

Unsigned Data Type Aliases:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Type | Size | Integer Range | Fractional Resolution | Effective Range\* |
| UQ4x4 | 8 bit (1 byte) | 0 – 15 | 1 / 16th | 0 to 16 |
| UQ8x8 | 16 bit (2 bytes) | 0 to 255 | 1 / 256th | 0 to 256 |
| UQ16x16 | 32 bit (4 bytes) | 0 to 65,355 | 1 / 65,536th | 0 to 65,356 |
| UQ32x32 | 64 bit (8 bytes) | 0 to 4,294,967,295 | 1 / 4,294,967,296th | 0 to 4,294,967,296 |
| UQ1x7 | 8 bit (1 byte) | 0 to 1 | 1 / 128th | 0 to 1 |
| UQ1x15 | 16 bit (2 byte) | 0 to 1 | 1 / 32,768th | 0 to 1 |
| UQ1x31 | 32 bit (4 byte) | 0 to 1 | 1 / 2,147,483,648th | 0 to 1 |
| UQ1x63 | 64-bit (8 byte) | 0 to 1 | 1 / 9,223,372,036,854,775,808th | 0 to 1 |

Signed Data Type Aliases:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Type | Size | Integer Range | Fractional Resolution | Effective Range\* |
| SQ3x4 | 8 bit (1 byte) | -15 to +15 | 1 / 16th | -16 to +16 |
| SQ7x8 | 16 bit (2 bytes) | -127 to +127 | 1 / 256th | -128 to +128 |
| SQ15x16 | 32 bit (4 bytes) | -32,767 to +32,767 | 1 / 65,536th | -32,768 to +32,768 |
| SQ31x32 | 64 bit (8 bytes) | -2,147,483,647 to  +2,147,483,647 | 1 / 4,294,967,296th | -2,147,483,648 to +2,147,483,648 |
| SQ1x6 | 8 bit (1 byte) | -1 to +1 | 1 / 64th | -2 to +2 |
| SQ1x14 | 16 bit (2 byte) | -1 to +1 | 1 / 16,384th | -2 to +2 |
| SQ1x30 | 32 bit (4 byte) | -1 to +1 | 1 / 1,073,741,824th | -2 to +2 |
| SQ1x62 | 64-bit (8 byte) | -1 to +1 | 1 / 4,611,686,018,427,387,904h | -2 to +2 |

\* Lower bound is inclusive, upper bound is exclusive