Math Concepts for Embedded Software Engineers.

**Look Up** **Tables**:

We have been taught in the schools the use of Math to solve real world problems especially in mechanical concepts like acceleration, velocity, etc. There are many embedded projects that use heavily the trigonometry functions like sin(x), cos(x). Sample projects like those that uses GPS navigation which tries to calculate the distance between points on a sphere(earth), it needs to compute a lot of trig functions in real time! These functions are terribly slow while executing on a microcontroller that runs on 8MHz.

These functions are usually computed using the Taylor series, which approximates most of complex function calculation like n! and sin (x), etc. using Taylor Series, sin(x) can be computed using the following series

sin(x) := x - x^3/3! + x^5/5! - x^7/7! +

As you see you need to calculate of terms like factorials, power, which is a disaster if you want to compute and calculate them on a microcontroller.

One solution that was used in the era of 80’s when computer game programmers used to work on 8 bit computers like Atari is to use a Lookup table, if you really must to work with trig functions, then Look-up tables are to rescue.

Look-up tables are precomputed values of some computation that you know you’ll perform during run-time. You simply compute all possible values at startup and then run the embedded software. For example, say you

needed the sine and cosine of the angles from 0-359 degrees. Computing them using

sin() and cos() would kill you if you used the floating-point processor, but with a look-up table your code will be able to compute sin() or cos() in a few cycles because it’s just a look-up. Here’s an example:

// storage for look up tables

float SIN\_LOOK[360];

float COS\_LOOK[360];

// create look-up table

for (int angle=0; angle < 360; angle++)

{

// convert angle to radians since math library uses

// rads instead of degrees

// remember there are 2\*pi rads in 360 degrees

float rad\_angle = angle \* (3.14159/180);

// fill in next entries in look-up tables

SIN\_LOOK[angle] = sin(rad\_angle);

COS\_LOOK[angle] = cos(rad\_angle);

} // end for angle

As an example of using the look-up table

for (int ang = 0; ang<360; ang++)

{

// compute the next point on circle

x\_pos = 10\*COS\_LOOK[ang];

y\_pos = 10\*SIN\_LOOK[ang];

// Do something with x\_pos, y\_pos

}

You actually first compute the precomputed values at initialization of the system, then in the main loop whenever you calculate sin or cos, you look up to their values in the look up table.

Yes, it takes a space overhead, but speed at ***run time*** is also important especially in low end microcontrollers like AVR.

Identifying Fast and Slow Operations

Optimizing your system to do its mathematical operations quickly requires you to understand

a bit more about your compiler and processor. Once you understand which

operations occur quickly (and which ones take up one line of code but compile to use

two libraries and an absurd amount of processing), you'll have the basis to optimize

your system.

So, addition and subtraction are fast. Shifting bits is fast. Division is very slow. Anything

with floating point is dead slow.

What about multiplication? On a DSP, it is fast: multiply and add together form a single

instruction (MAC for multiply-accumulate). On a non-DSP (e.g. an ARM or your PC),

multiplication is between addition and division, closer to addition

for (i=0; i<100; i++)

if ((i%10) == 0) {

printf(“%d percent done.”, i);

}

}

for (i=0; i<100; i++)

if (i & 7) { // this will print out every 8th pass

printf(“%d percent done.”, i);

}

}

**Fixed Point Math**

Some microcontrollers don’t support Floating point operations, as they don’t have an FPU. Most operations on these microcontrollers are simulated in software. You can look at your map files of your specific microcontroller to see how much code space that a floating point operation between the sum of two floating points took how much a big a space at the end. In addition to code space increase, floating point operation are slow, floating point operations are expensive in embedded software. Because they are emulated in Software which takes a lot of CPU Cycles to emulate the mathematical operations of floating points. The idea to the following explanation is to fake Floating point numbers with Fixed Point Math. The point is to fake floating point with integers.

Implementation-wise, there are two main ways to do fixed point math: CPU or dedicated logic.

For the CPU approach, it can be a full-blown DSP (Digital Signal Processor), or it can be an application processor running at a high clock rate or with built-in support for SIMD instructions. Dedicated logic can be implemented through a FPGA, or it can be a hardware accelerator that becomes part of an SoC (System on Chip). Each approach has its pros and cons. And because there are two approaches for doing fixed point math, the code snippets presented in this chapter will also be in one of these two forms: C/C++ or Verilog/System Verilog.

In case of AVR's and PIC's compiler knows that there is no fpu available, so it will translate every single operation to a bunch of commands that CPU supports. It will have to normalize both operands to a common exponent, then perform operation on mantissa like on integral numbers, then adjust exponent. This is quite a lot of operations so emulated floating point is slow. And, beside that, if you optimize for size, every floating point operation may become a function call.

And on ARM arch things may be quite weird. There are ARM's with FPU and without. And you may want to have universal application which will run on both. In such case there is a tricky (and slow) scheme. Application uses FPU commands. If your CPU does not have FPU, then such command will trigger an interrupt and in it OS will emulate the instruction, clear error bit and return control to an application. But that scheme occurred to be very slow an is not commonly use

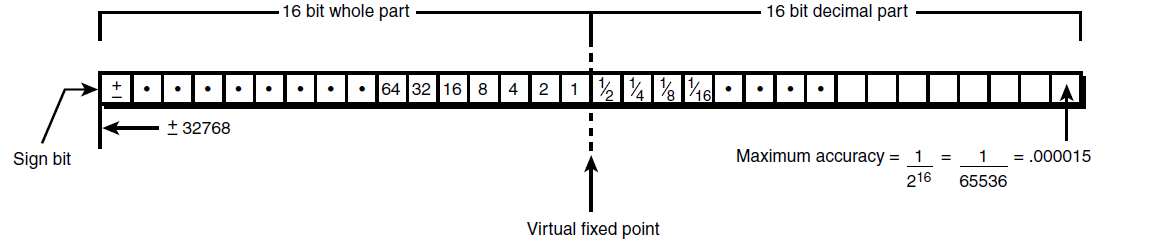
Floating-point is just scientific notation in base-2. Both the mantissa and exponent are integers, and softfloat libraries will break up floating-part operations into operations that affect the mantissa and exponent, which can use the CPU integer support.

For example, (x 2n) \* (y 2m) = x \* y 2n+m.

[Fixed-Point vs. Floating-Point Digital Signal Processing | Education | Analog Devices](https://www.analog.com/en/education/education-library/articles/fixed-point-vs-floating-point-dsp.html)

Remember at school we can have a number like 10.7. but can we represent it as an integer? We can’t we don’t have a decimal place in integer values, but we can scale it up by 10 and the result would be 107. Which is an integer. You scale numbers by some factor and that make sure to take this scale into consideration when doing mathematics.

We will consider using 32bit integers for our floating point representation. There are many formats of Fixed point math that depends on the size of the integer that has been used. We will show an example of a 16.16 fixed point math.



You put the whole part in the upper 16 bits and the decimal part in the lower 16 bits. Hence, you’re scaling all numbers by 2^16, or 65,536. Moreover, to extract the integer portion of a fixed-point number, you shift and mask the upper 16 bits, and to get to the decimal portion, you shift and mask the lower 16 bits. Here’s some working types for fixed-point math:

#define FP\_SHIFT 16 // shifts to produce a fixed-point number

#define FP\_SCALE 65536 // scaling factor

typedef int FIXPOINT;

Here’s a macro that converts an integer to fixed-point:

#define INT\_TO\_FIXP(n) (FIXPOINT((n << FP\_SHIFT)))

For example:

FIXPOINT speed = INT\_TO\_FIXP(100);

And here’s a macro to convert floating-point numbers to fixed-point:

#define FLOAT\_TO\_FIXP(n) (FIXPOINT((float)n \* FP\_SCALE))

For example:

FIXPOINT speed = FLOAT\_TO\_FIXP(100.5);

Extracting a fixed-point number is simple too. Here’s a macro to get the integral portion

in the upper 16 bits:

#define FIXP\_INT\_PART(n) (n >> 16)

And to get the decimal portion in the lower 16 bits, you simply need to mask the integral

part:

#define FIXP\_DEC\_PART(n) (n & 0x0000ffff)

**Addition and Subtraction**

Addition and subtraction of fixed-point numbers is trivial. You can use the standard +

and – operators:

FIXPOINT f1 = FLOAT\_TO\_FIX(10.5),

f2 = FLOAT\_TO\_FIX(-2.6),

f3 = 0; // zero is 0 no matter what baby

// to add them

f3 = f1 + f2;

// to subtract them

f3 = f1 – f2

**Multiplication and Division**

Multiplication and division are a little more complex than addition and subtraction. The problem is that the fixed-point numbers are scaled; when you multiply them, you not only multiply the fixed-point numbers but also the scaling factors. Take a look:

f1 = n1 \* scale

f2 = n2 \* scale

f3 = f1 \* f2 = (n1 \* scale) \* (n2 \* scale) = n1\*n2\*scale^2

See the extra factor of scale? To remedy this, you need to divide or shift out the one

factor of scale^2. Hence, here’s how to multiply two fixed-point numbers:

#define FP\_MUL (f1,f2) ((f2\*f1)>>FP\_SHIFT)

f3 = ((f1 \* f2) >> FP\_SHIFT);

Division of fixed-point numbers has the same scaling problem as multiplication, but

in the opposite sense. Take a look at this math:

f1 = n1 \* scale

f2 = n2 \* scale

Given this, then

f3 = f1/f2 = (n1\*scale) / (n2\*scale) = n1/n2 // no scale!

Note that you’ve lost the scale factor and thus turned the quotient into a non-fixedpoint

number. This is useful in some cases, but to maintain the fixed-point property,

you must prescale the numerator like this:

f3 = (f1 << FP\_SHIFT) / f2;

The problem with both multiplication and division is overflow and underflow. In the case of multiplication, the result might be 64-bit in the worst case. Similarly, in the case of division, the upper 16 bits of the numerator are always lost, leaving only the decimal portion. The solution?

Use a 24.8-bit format or use full 64-bit math.

This will allow multiplication and division to work better because you won’t lose everything all the time, but your accuracy will fall apart.

FIRMWARE CONCEPTS IN C

**Stack painting**

An effective way to measure the amount of stack space needed consists of filling the

estimated stack space with a well-known pattern. This mechanism, informally referred to as

stack painting, reveals the maximum expansion of the execution stack at any time. By

running the software with a painted stack, it is in fact possible to measure the amount of

stack used by looking for the last recognizable pattern, and assuming that the stack pointer

has moved during the execution at most until that point.

We can perform stack painting manually in the reset handler, during memory initialization.

To do so, we need to assign an area to paint. In this case it would be the last 8 KB of

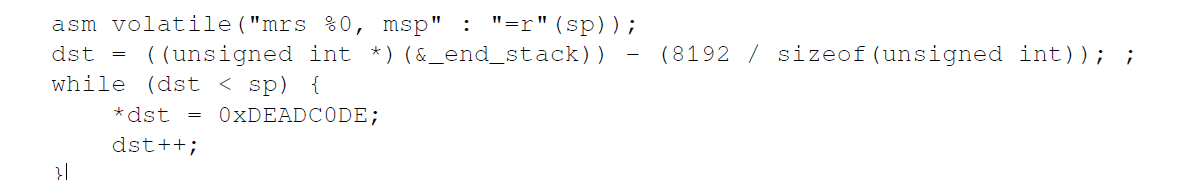
memory up until \_end\_stack. Once again, while manipulating the stack in the

reset\_handler function, local variables should not be used. The handler

function will store the value of the current stack pointer into the global variable sp:

static unsigned int sp;

Within the handler, the following section can be added before invoking main:



The first assembly instruction is used to store the current value of the stack pointer to the

variable sp, to ensure that the painting stops after painting the area, but only up until the

last unused address in the stack:



The current stack usage can be checked periodically at runtime, for instance in the main

loop, to detect the area painted with the recognizable pattern. The areas that are still

painted have never been used by the execution stack so far, and indicate the amount of

stack still available.

This mechanism may be used to verify the amount of stack space required by the

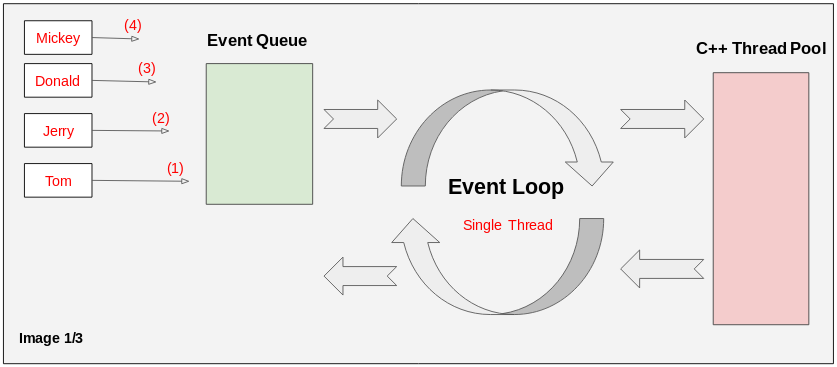
application to run comfortably. According to the design, this information can be used later

on to set a safe lower limit on the segment that can be used for the stack.

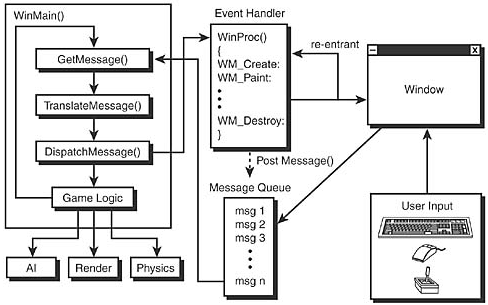
REAL TIME DESIGN PATTERNS

EVENT BASED SYSTEM

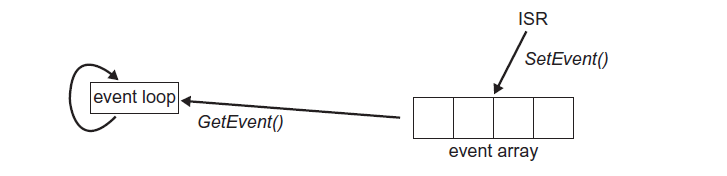
Remember Windows Event Loop?



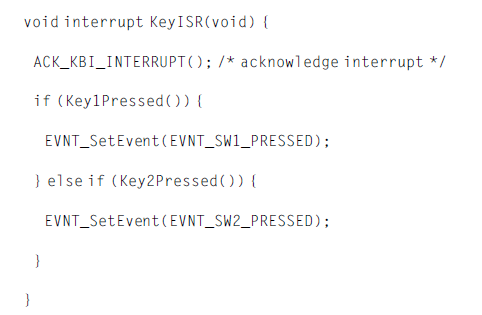
Taken from internet ( JAVA Script Event LOOP)

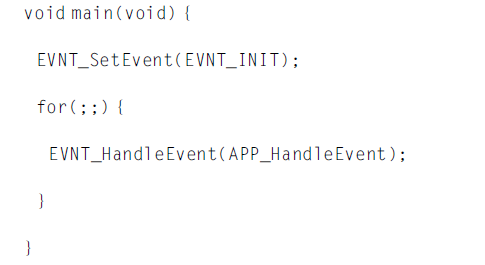


Windows Messaging in Games ☺ from Tricks of the Windows Game Programming Gurus Andre’ LaMothe <3









TRIGGERS

So far we have the ability and infrastructure to flag an event and to process it in the main

loop. What is missing is a way to do something in a time-triggered fashion: for example to

blink an LED every second, or to turn on an LED 500 ms after a button has been pressed.

For this we are going to introduce the concept of a trigger. Triggers are sometimes also

used to denote a hardware functionality: for example a microcontroller hardware is set up to

trigger on a read or write access to halt the processor in order to implement what is also

known as watchpoint. We are using triggers here in a slightly different way. We want the

application to trigger at a given time in the future.

