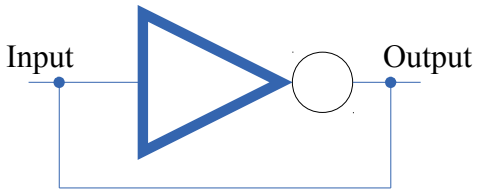
Timing

Clock signals

Microcontrollers are often required to measure time intervals and to generate events (or signals) at a particular rate. Typical applications include music generation, flow meters , speedometers , time of day clocks etc. If applications such as these are to function correctly then an accurate measure of time is required. How can this be achieved? The first ingredient is a stable, predictable clock signal. ***In the digital world, a clock signal is one that switches from low to high and back again at a particular rate.***

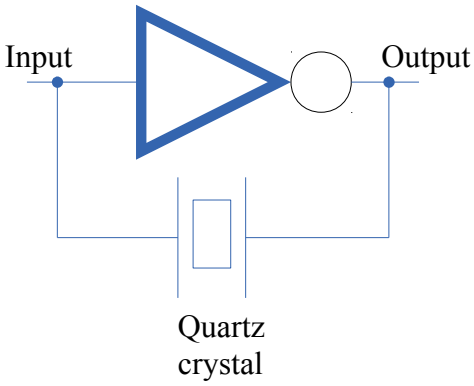
A simple clock signal can be generated using a NOT gate as follows:



This works as follows:
Assuming the input is 0 to start with...
The output will be the opposite of the input: **1**
The output feeds back to the input : 1
The output will be the opposite of the input: **0**
The output feeds back to the input : 0
The output will be the opposite of the input: **1**
The output feeds back to the input : 1
The output will be the opposite of the input: **0**

:
:

As you can see, the output will switch between high and low – but at what rate? Well, this sort of circuit has a very unpredictable switching rate. The switching rate can be stabilized by adding additional components as shown below:

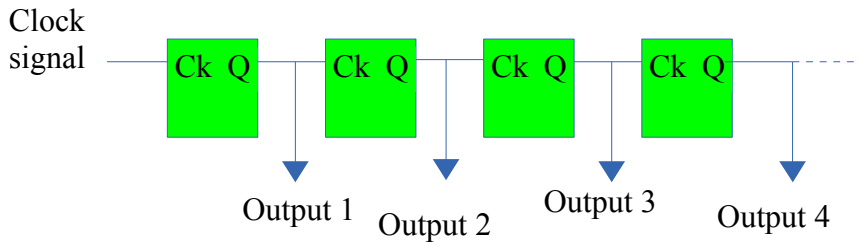


The added ingredient is a quartz crystal that is designed by a manufacturer resonate at a particular frequency. The oscillator is forced to "lock in" to the resonant frequency of the crystal which is a known and stable frequency. If you look at digital electronic circuit boards you will often see little silver cans as shown below. These contain quartz crystals. Top and bottom views of an 8MHz quartz crystal "can" are shown above. For the purposes of this note you think of a quartz crystal as a miniature tuning fork that resonates at a very high frequency.

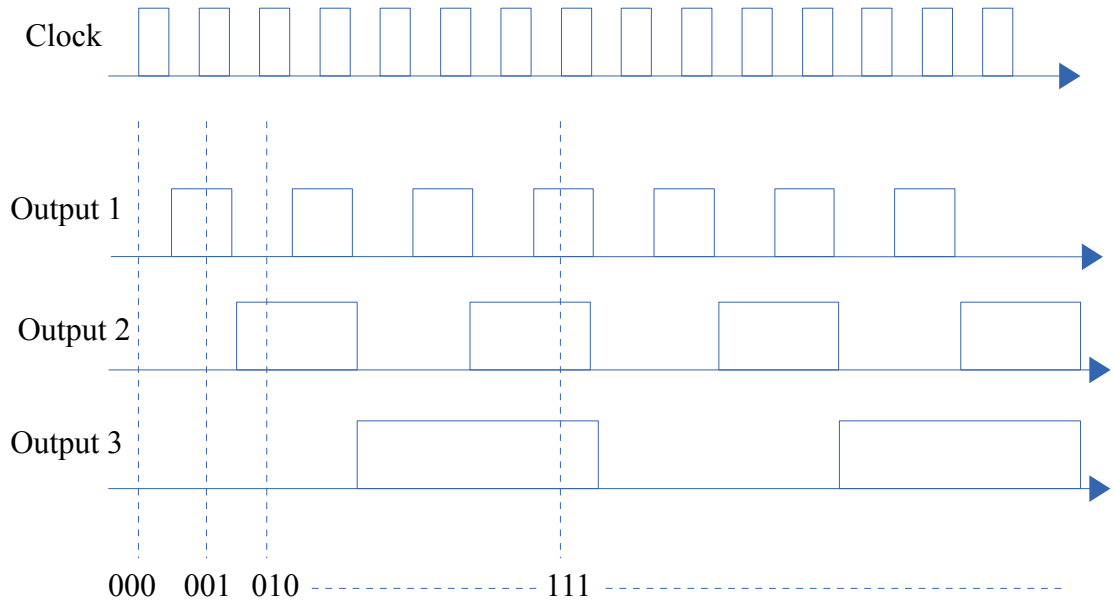
Not all microprocessor systems make use of quartz crystal oscillators. Some make use of oscillator circuits within the microprocessor that are factory calibrated to oscillate at a particular frequency. While these are not as accurate as their quartz equivalents, they are significantly cheaper to use and are often good enough.

Counting

Having established a stable clock signal, the next ingredient required is a digital counter. A counter is typically made up from a chain of digital latching circuits called flip-flops.



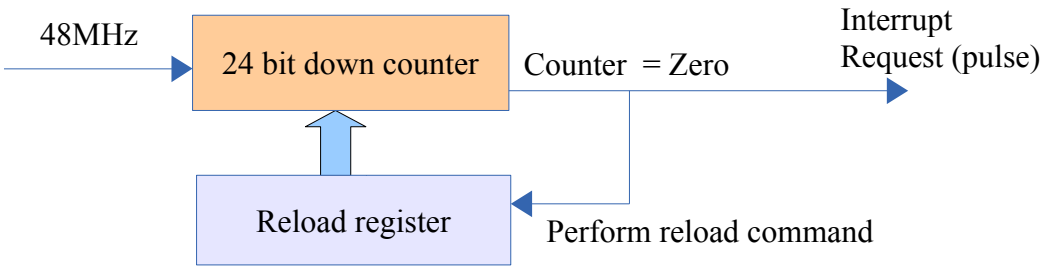
The incoming clock signal passes through the chain of flip flops. At each stage, the frequency is halved so, Output1 changes state twice as quickly as Output 2 etc. Graphically, this can be represented as follows:



If we look vertically along the waveforms we can see that the device is in fact counting (in binary). Timers in microcontrollers typically have counters with 16 bit count registers (16 flip flops). Counters can be made to count up as well as down. They can also be used in circuits that compare outputs to particular values. If the count matches the value an event is generated. This allows us to do thinks like generate music and control the speed of a motor. It also allows us to generate regular, periodic **interrupts**.

The Cortex M SysTick interrupt.

All ARM Cortex M microcontrollers have a 24 bit counter called the SysTick counter. This counter counts down from a user defined value. When it reaches zero it can generate an interrupt request. ***The interrupt request is equivalent to a hardware generated function call***. In other words, your program is busy doing some task when, all of a sudden, an internal hardware system pauses the current operation and goes and executes a different function in your program. When the function is complete, the interrupted process continues.



Example:
An application requires a program to periodically check an input signal. The signal must be checked every millisecond (i.e. 1000 times per second). Use the SysTick system to implement this periodic checking.

In order to solve this problem we need to know the input clock frequency. In the case of our LPC1114, the input frequency is 48MHz. What do we divide this by to get 1000Hz? Well, the answer is 48000. If we put this number into the Reload register and start the SysTick system then an interrupt request will be generated every 48000 clock pulses. Each clock pulse occurs every $1/48000000^{th}$ of a second so our interrupt request occurs very 1000^{th} of a second. (Strictly speaking we should put the number 48000-1 into the reload register). Our code to check the input signal might be something like this:

```
void SysTick()
{
        if (GPIO0DATA & BIT4)
        {
                // BIT4 set so respond in some way.
        }
}
```

As you can see, this is just a normal C function.
How does the SysTick system know that it must call this function and not some other? There is a table in memory called the **Interrupt Vector Table**. Each entry in the table contains the address of the function that should be called if a particular interrupt is generated. If we put the address of the

SysTick function into the correct slot of the table then it will be called when a SysTick interrupt request is generated.

The SysTick interrupt is intended to be used as a periodic interrupt source. Multitasking embedded operating systems use it to manage task switching. Simpler systems can use it for a time of day clock or for scanning inputs and refreshing outputs in a periodic, predictable manner.

Using timers to generate waveforms.

It is possible to use hardware timers to produce periodic waveforms. These waveforms can be used for a number of applications including communications, motor control and music generation. The system works as follows: A counter counts an input clock signal. It is constantly compared with two other values ("Compare 1" and "Compare 2" registers below). If the counter value matches one of the registers (Compare 1), it is reset to zero. At the same time, an flip-flop is set high. If the counter matches the second register (Compare 2), the flip-flop is set low. If we vary the contents of the "Compare 1" register, we can vary the flip-flop output frequency. If we vary the contents of "Compare 2" we can vary the percentage of the time the output of the flip-flop is high. The flip-flop output can be used to drive a number of circuits include motor controllers or simply a buzzer. Varying the values in Compare 1 and 2 allows us to produce output tones with little CPU overhead. Musical birthday cards work like this.