Netduino for beginners

A gentle introduction using basic electronic projects

Georgi Hadzhigeorgiev

v1.11

Table of Contents

Introduction

Hello!

I presume you have some very basic electronics and programming knowledge and, seeing all the amazing stuff you can do with Netduino, you bought one. You probably followed a few examples in articles that you found and tried a few things, but now want to do some stuff yourself. Perhaps you're a software developer and don't have much experience in electronics, or you're in exactly the opposite position and know how to design electronic devices, but haven't much experience in programming. You're looking for some introductory stuff coming from either way. Well then, this may be the right material to read.

I started writing this as a basic article for a "Hello World!" blinking LED example but, in comparison to other similar projects, I wanted to show people without an electronics background how to design simple schematics themselves. My other aim was to provide readers with a good understanding of electronics principles and processes, thus boosting their confidence to turn their dream devices into something real. Later, with great support and encouragement from the Netduino community, I expanded the material with more complicated examples, including the introduction of some programming techniques for beginners.

I hope everyone will find and learn something new and interesting in this material. If so, then it fulfils its purpose.

Regards,

Georgi Hadzhigeorgiev

I want to thank my colleagues John Kerr, from whom I received the most valuable technical feedback and Andrew O'Hara, who helped me to improve the quality and readability of the text.

Keeping with the tradition of special family thanks, I want to thank my wife Maria, for her support, as well as being the reason I started writing this material, and to my daughter Yoana, for being patient while daddy spent nights in front of the computer instead of playing with her.

Introduction to electronics principles and components

Initially I had no plans to include any such information, but some readers pointed out that it would be good for people without any knowledge of electronics whatsoever to be able to get an introduction to elementary electronics basics and principles. This is a huge area and I spent some time deciding what was to be included here. In the end, I decided to provide readers with information about a few laws and principles, and to explore some basics of how the electronic components work that we're going to use in our projects. You'll find this section full of links to materials that you can read. I suggest you read these materials when you have time but not all of the material's content is directly relevant to our needs, so I kept it out of the current content. Links helped me keep this material short, as I'll only provide things here that we'll need or that require further explanation.

Electric potential, voltage and current

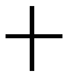
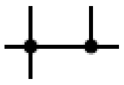
It's difficult to find a nice, easy way to explain electric potential, so I ended up simply pointing you to the Wikipedia page. As it states: "electric potential … at particular point in space is the electric potential energy divided by charge associated with a static electric field". You have to read the entire article and dig into the related links if you want to understand the true nature of potential. Fortunately, we have sort of a shortcut to all of this and from now on, we will understand the potential simply as a *difference*. That difference has a scalar measurement and its name is voltage. When two points are connected, using some conductive medium (such as wire, electrolyte, etc.), the voltage (the difference between point potentials) makes electric charges (electrons and/or ions) flow between them. This flow we call Current.

Conductors, insulators and semiconductors

Depending on the conductive medium's behaviour, we can categorise materials such as Conductors, Insulators and Semiconductors. Based on those materials sing those materials and their properties, various electronic components exist. I'd recommend exploring this list as the more components you are familiar with, the easier you'll find their application in your electronic schematics design, thus making it easier, more functional and more enjoyable. From this list of components, we'll mostly be interested in the following:

Wires and wiring

If you want to transfer electricity from one point to another then you need wire. There are many types of wires, grouped by type of material, size, etc. The most important characteristic that wires have is that they can conduct electricity with minimum resistance. This is very important because we want to minimise any loss of electricity (energy) during the transfer. Ideally, wires should have no resistance at all, but we don't live in a perfect world, so they do have some. Normally, for the purposes of home build electronic projects, this resistance isn't enough to cause any significant problems, so it's usually ignored thus simplifying the calculations we have to do. There are situations where we can't ignore this resistance, and sometimes there are situations when we even benefit from it. For this, if there is no specific note made about wire resistance, then one should presume it's unimportant for the purposes of the schematics. Here you can have a look at the details of different types of conductors used to make wires.

In schematics, lines connecting elements represent wires. Usually, when wires aren't connected, they look like this . When they're connected, they look like this .

Resistor

Like wires, resistors are made from conductors but with higher resistance. This is usually their primary function. You may wonder why we'll possibly need to put resistance in the way of the electricity when transporting it.

One simple example of a reason is to limit the current in the circuit. Imagine we have a 5V source of electricity and we need to power a load that has internal resistance of 20 Ohms, but which cannot sustain more than 10mA current. Later we'll become more familiar with Ohm's law, but now we have to mention at least the main relationship between the voltage, resistance and the current flowing through it: I = V/R. From this equation, we can see that a higher voltage over the same resistance will result in a higher current. In addition, a higher resistance will result in a lower current. This also means that if a particular current flows through resistance then it will cause a voltage to appear on either side of the resistance. This voltage we call "voltage drop". So in our imaginary case, if we connect the load directly to the voltage source, then the current that will flow through the load will be 5V / 20Ω = 0.25A (250mA). This is something we have to prevent and using a resistor is one of the possible ways. Without going into too much detail, to limit the current in the circuit to 10mA we have to use a resistor with value of (5V / 0.01A) – 20 = 480Ω. Now let's quickly verify that in a clearer way (I presume you are a beginner and as such, upper calculation may look odd to you): 5V / (480Ω +20Ω) = 10mA.

This limiting of the current isn't without a cost, of course. Current flowing through the resistor cases a voltage drop across the resistor and it may be significant. In the previous example the voltage drop will be 480Ω x 0.01A = 4.8V! This means the load will get only 5V – 4.8V = 0.2V and unless this is enough voltage, it may never operate normally. Therefore, this was an illustrative rather than practical example. In Project One in this tutorial, we will see how a resistor is used to power a LED. You can try to do the calculations yourself before reading the tutorial. For this, you need to know that a LED requires 2V and no more than 20mA to operate properly. Voltage source is 5V. Now do the maths.

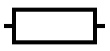
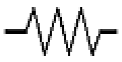
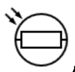
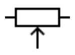
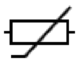
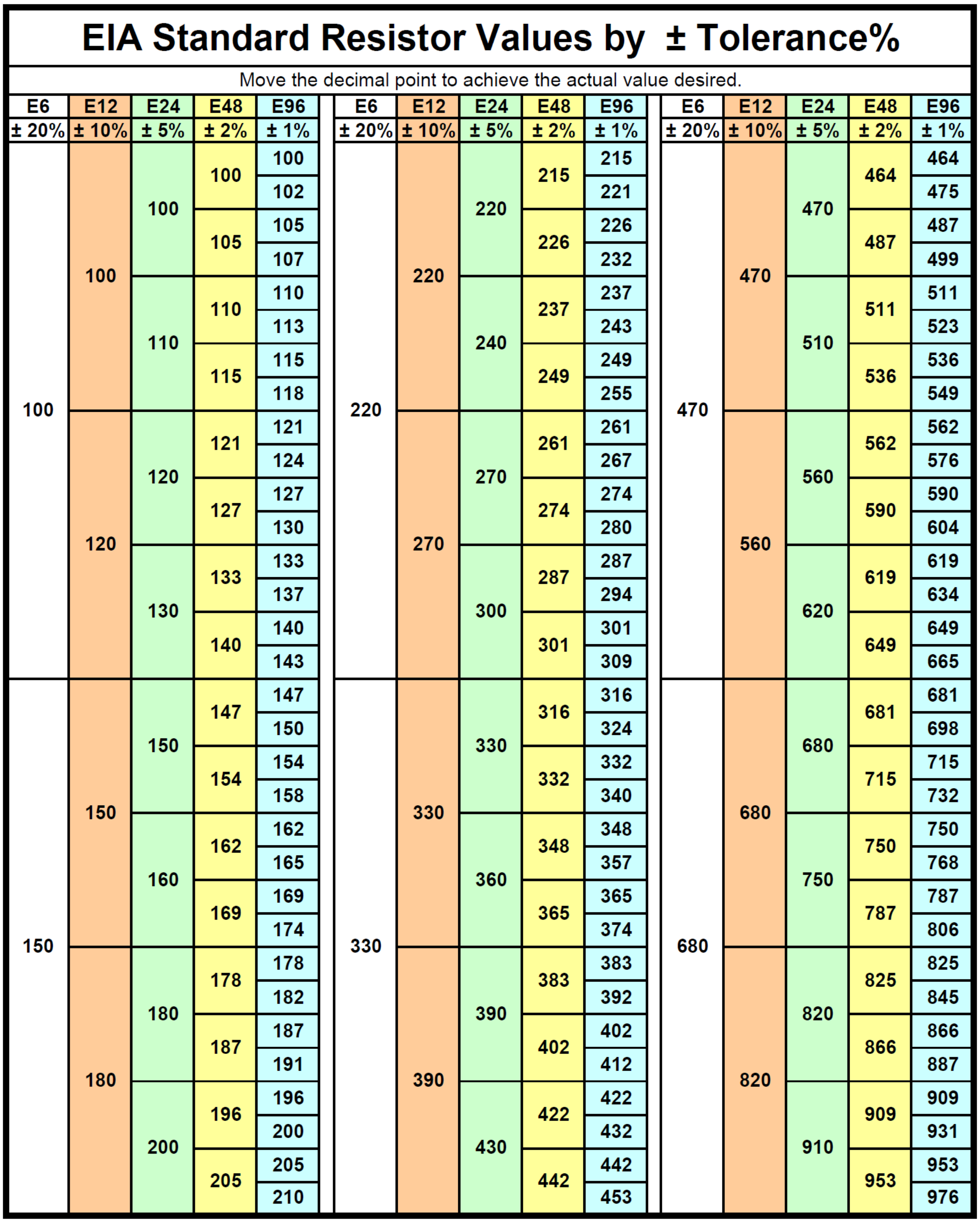
In texts and schematics, the letter R is used to indicate a resistor and is usually followed by an index (R1, R_1, Rc, R_{BE} etc.) in order to specify an exact element where there is more than one resistor on the schematics. Graphical representation in schematics may vary but most standard ones are  and  Resistor variants have an additional graphical sign, added over the rectangle or zigzag, which represent the resistor's speciality like LDR , potentiometer , thermistor  etc.

Table of standard resistor values



EIA Standard Resistor Values by ± Tolerance%

Move the decimal point to achieve the actual value desired.

E6	E12	E24	E48	E96	E6	E12	E24	E48	E96	E6	E12	E24	E48	E96
± 20%	± 10%	± 5%	± 2%	± 1%	± 20%	± 10%	± 5%	± 2%	± 1%	± 20%	± 10%	± 5%	± 2%	± 1%
100	100	100	100	100	220	220	220	215	215	470	470	470	464	464
				102					221					475
			105	105				226	226				487	487
				107					232					499
		110	110	110			240	237	237			510	511	511
				113					243					523
			115	115				249	249				536	536
				118					255					549
	120	120	121	121		270	270	261	261		560	560	562	562
				124					267					576
			127	127				274	274				590	590
				130					280					604
		130	133	133			300	287	287			620	619	619
				137					294					634
			140	140				301	301				649	649
				143					309					665
150	150	150	147	147	330	330	330	316	316	680	680	680	681	681
				150					324					698
			154	154				332	332				715	715
				158					340					732
		160	162	162			360	348	348			750	750	750
				165					357					768
			169	169				365	365				787	787
				174					374					806
	180	180	178	178		390	390	383	383		820	820	825	825
				182					392					845
			187	187				402	402				866	866
				191					412					887
		200	196	196			430	422	422			910	909	909
				200					432					931
			205	205				442	442				953	953
				210					453					976

Inductance and coil

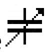
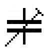
Capacitance and capacitor

Capacitors (also called condensers) are like rechargeable batteries: they store electric charge. Unlike them, however, they can only be charged and discharged very quickly (at least compared to batteries), so they can only store small amounts of electrical charge. In addition, they're passive rather than active elements. There are many types of capacitors but in its simplest form, you can think of the capacitor as a device constructed from two conductive plates with insulator between them. How much charge will be stored (accumulated) depends on the size of those plates and the thickness of the insulator between them. That capacitance (ability to store charge) is measured in Farads (F). Higher capacitance means more charge can be stored.

The process of storing or depleting the charge isn't instantaneous and takes some time. It takes more time to fill higher capacitance and more time to deplete it. This effect can be increased by attaching a resistor to the capacitor. Through the process of selecting a resistor value, we can control the charging current, thus controlling charging and depletion time. On this page, you can read more about this type of Resistor-Capacitor relationship. Similar to RC circuits there are LC circuits as well as RLC circuits. In these circuits, we can replace static value capacitors with variable ones, or their miniature variant called trimmer capacitors, thus allowing us to vary the frequencies. You should also familiarise yourself with decoupling (bypass) capacitor.

One practical usage example you can find is in "Third approach – Using NAND gate (NOT AND logic)" part of this tutorial, where we will use a 0.1µF ceramic capacitor to assure the load receives stable power during logic switching between levels.

The above mention of ceramic capacitors means that there are other types. There are indeed, but they're classified by a more important parameter: polarisation. Based on that, there are two main types of capacitors: polarised and non-polarised. The biggest difference between these types is that polarised capacitors must be connected correctly, as they have distinctive positive and negative plates. Swapping the polarisation will damage the capacitor and may cause an explosion. Unless you want to make little fireworks, you should never allow reverse polarisation on this type of capacitors. On the other hand, material classification is important for the job that we'll appoint the capacitor to do. For example, ceramic capacitors are "quicker" and have low inductance compared to electrolytic ones, which on the plus side have the advantage of higher capacitance, thus making them more suitable for supplying higher currents at lower frequencies. You can find more information about the various types of capacitors and their application on this page.

In texts and schematics, the letter C is used to indicate a capacitor and is usually followed by an index (C1, C_1, C_E, C_{BE} etc.) in order to specify an exact element where there is more than one capacitor on the schematics. Graphical representation in schematics may vary but most used are ⊣⊢ for non-polarised capacitors and ⊣⊏ for polarised ones. Variable capacitors have an additional arrow drawn over the plates, for which the symbol looks like , with  representing a trimmer capacitor.

Most commonly used capacitor values

pF	pF	pF	pF	µF	µF	µF	µF	µF	µF	µF
1.0	10	100	1000	0.01	0.1	1.0	10	100	1000	10,000
1.1	11	110	1100							
1.2	12	120	1200							
1.3	13	130	1300							
1.5	15	150	1500	0.015	0.15	1.5	15	150	1500	
1.6	16	160	1600							
1.8	18	180	1800							
2.0	20	200	2000							
2.2	22	220	2200	0.022	0.22	2.2	22	220	2200	
2.4	24	240	2400							
2.7	27	270	2700							
3.0	30	300	3000							
3.3	33	330	3300	0.033	0.33	3.3	33	330	3300	
3.6	36	360	3600							
3.9	39	390	3900							
4.3	43	430	4300							
4.7	47	470	4700	0.047	0.47	4.7	47	470	4700	
5.1	51	510	5100							
5.6	56	560	5600							
6.2	62	620	6200							
6.8	68	680	6800	0.068	0.68	6.8	68	680	6800	
7.5	75	750	7500							
8.2	82	820	8200							
9.1	91	910	9100							

The most common working voltages (DC), classified according to Capacitor type

Ceramic	Electrolytic	Tantalum	Mylar (Polyester)	Mylar (Metal Film)
	10V	10V		
16V	16V	16V		
		20V		
25V	25V	25V		
	35V	35V		
50V	50V	50V	50V	
	63V			
100V	100V		100V	
	160V			
			200V	
	250V			250V
	350V			
			400V	
	450V			
600V				
			630V	
1000V				

P-N junction

Diode

LED

It’s great if you have access to the manufacturer’s data but usually you don’t. That’s not a big problem as you can use the following specs:

Type	Colour	$I_{F\ max}$	$V_{F\ typ}$	$V_{F\ max}$	$V_{R\ max}$	Luminous intensity	Viewing angle	Wavelength
Standard	**Red**	30mA	1.7V	2.1V	5V	50mcd @ 10mA	60°	660nm
Standard	**Bright red**	30mA	2.0V	2.5V	5V	80mcd @ 10mA	60°	625nm
Standard	**Yellow**	30mA	2.1V	2.5V	5V	32mcd @ 10mA	60°	590nm
Standard	**Green**	25mA	2.2V	2.5V	5V	32mcd @ 10mA	60°	565nm
High intensity	**Blue**	30mA	4.5V	5.5V	5V	60mcd @ 20mA	50°	430nm
Super bright	**Red**	30mA	1.85V	2.5V	5V	500mcd @ 20mA	60°	660nm
Low current	**Red**	30mA	1.7V	2.0V	5V	5mcd @ 2mA	60°	625nm

Where

$I_{F\ max}$	Maximum forward current, forward just means with the LED connected correctly
$V_{F\ typ}$	Typical forward voltage, V_L in the LED resistor calculation. This is about 2V, except for blue and white LEDs for which it is about 4V
$V_{F\ max}$	Maximum forward voltage
$V_{R\ max}$	Maximum reverse voltage. You can ignore this for LEDs connected the correct way round
Luminous intensity	Brightness of the LED at the given current measured in mcd = millicandela
Viewing angle	Standard LEDs have a viewing angle of 60°, others emit a narrower beam of about 30°
Wavelength	The peak wavelength of the light emitted, this determines the colour of the LED measured in nm = nanometre

There’s one design problem you should be aware of when you use a battery instead of a DC power supply. In situations like these, the battery voltage drops and the LED emits no light. You check the battery and it’s still OK, but the LED refuses to light. Well, the problem is that you designed the schematics too precisely☺. Hmm, how can precision be a bad thing? Let me explain.

Let’s take an example where you’re powering, through a current limiting resistor, 2 standard red LEDs connected in series, with a standard CR2032 3V battery, which has 240mAh before it drops down to 2V. Based on general LED specifications, you calculated the resistor value: R = (3V – 2V)/20mA = 50Ω and you’ve chosen R = 47Ω from standard resistor values (we’ll ignore the tolerance, as it’s unimportant in illustrating the problem). Right, what we have now is I_L = (3V – 2V) / 47Ω = 21mA, which should make the LED bright and shiny enough for approximately 8 hours. Soon, the battery drops down to 2.5V. Let’s see what we have now for I_L = (2.5V – 2V) / 47Ω = 10mA, which is twice lower than initially desired. It may still be visible but things aren’t going very well for us. Further, battery voltage goes down to 2.1V. It should still be enough to power the LED, but now the current limiting resistor, which has been our friend before, becomes our enemy: I_L = (2.1V – 2V) / 47Ω = 2mA and the LED will be dead, when we still have enough battery energy. Let's look at the voltage drop across the resistor in all those cases. Initially it was 1V, then 0.5V then 0.1V. This voltage is called "*head voltage*". This is because it's kind of an indication of how much voltage in reserve we have before the LED will stop emitting light. Obviously a higher value indicates a larger reserve. If we power the same schematic with a 4.5V battery, we'd have 1.5V in reserve and we'll still consume 20mA (imagining we use an exact 125Ω resistor). In increasing the battery voltage we'll have a greater reserve, thus we can guarantee a longer working time and fewer battery changes.

Back to the precision problem: if you are too precise (you may have used the minimum required voltage, the precise resistor value, etc.) then things will work exactly as expected… but for a very short time☺. Don't forget to look at the schematics you design from a practical point of view.

PUT

Transistor

Thyristor

Like transistors, Thyristors have the ability to be switched ON by a controlling current (applied to their Gate) and tend to stay ON until the current flowing through their controlled circuit drops below a certain level. The advantage of thyristors is that they can control much higher voltages and currents compare to transistors. There is a wide variety of different types of thyristors like SCR, DIAC, TRIAC, GTO, etc. Each one of these types exists to solve a different type of problem. We'll use some of these types in future projects but will go into the particular details when the time comes.

Fundamental circuit laws and analysis

Ohm's law

Kirchhoff's laws

Methods for analysis

Digital logic

Basic functions

Here you can find basic information about logic gates.

NOT gate

AND gate

NAND gate

Diode logic (DL)

Resistor-Transistor logic (RTL)

Diode-Transistor logic (DTL)

Transistor-Transistor logic (TTL)

Emitter-Coupled logic (ECL)

Integrated circuit

IC is ....

Microprocessors

Microprocessor is ....

Microcontrollers

Microcontroller is ....

Mechanical elements

Switch

Relay

Introduction to Netduino

Netduino can blink … a LED

Part I (electronic)

This is kind of a widely accepted “Hello world” project when it comes to Netduino. Why? Because at first glance it’s very simple, but there are a few points where an absolute beginner may fail and as a result can “burn” their Netduino. This would be too bad and we want to prevent it from happening, so here’s how.

In this project, we’re going to use Netduino to power up and down (flash) a Light-Emitting Diode (LED). LEDs come in different shapes, sizes and colours, but what they do is emit light (what a surprise☺). This is a good source of information about them, and so is this. From all the information available about LEDs, we need only those bits that will affect our calculations and actual assembly. These are:

1. Long leg is the Anode (+). We need this information when assembling the board.
2. If we know the manufacturer’s specs for the LED, that’s good, but if not, we can make a good guess based on the colour of the LED we’re powering, as specified here.

So, for our first project, let’s select a standard red LED. Based on that, and the information we have about LEDs, we know we have to apply 2V and no more than 20mA to it so it will be bright enough without being damaging. We know that it can take a bit more voltage and a few more milliamps but we don’t want to push it to the limit. In addition, there is a production tolerance which could be either positive (+) or negative (–), so it’s better to assume lower values just to be on the safe side.

So far, so good and we can move on the next step, which is to figure out what the current limiting resistor will be. We need this resistor so that when we connect the LED to a power source it will not burn. Without the resistor, the LED will be damaged even if the source is 2V. Why? Simply because the LED’s internal resistance is very low and the current will be too high. Right, we have one value – 20mA – but to find out the resistance, we need a voltage. At that point, we have to think about the general schematics.

First, we’ll look at the Netduino specs. According to these, digital output pins can give us 3.3V and 8mA (pins 2, 3 and 7 can give us 16mA) while analogue output pins can give us only 2mA. In total, the controller cannot power anything consuming more than 200mA without being damaged. Well, considering the voltage (3.3V) this should be enough to power the LED, but an 8mA current could be far below the brightness we want from the LED. We are also going to exclude 16mA pins, as in a future expansion of this project we'll want to use at least 4 LEDs, and Netduino has only 3 x 16mA pins. So what can we do? We can use another power source, like another power supply that can provide sufficient voltage and current. Alternatively, we can use the 5VDC output on Netduino itself. The voltage is sufficient and we can consume up to 800mA with an external power supply and 500mA when powered through USB (assuming we'll have 4 LEDs to power, this makes 4x20mA = 80mA, which is far below those limits). We'll use the 5VDC but we need somehow to "trigger" the LED on and off. How can we do this? Let's use a transistor as a switch. Before we dig into the schematics, let's have a look at basic transistor parameters and working modes.

Parameter	Definition and description
Type number	Type number of the device is an individual part number given to the device. Device numbers normally conform to the JEDEC (American), Pro-Electron (European) numbering systems or Japanese standard system.
Case	Case style - a variety of case standard case styles are available. These normally are of the form TOxx leaded devices and SOTxxx for surface mount devices. Note it is also important to check the pin connections, as they are not always standard. Some transistor types may have their connections in the format EBC whereas occasionally they can be ECB, and this can cause confusion in some cases
Material	Material used for the device is important as it affects the junction forward bias and other characteristics. The most common materials used for bipolar transistors are silicon and germanium
Polarity	Polarity of the device is important. It defines the polarity of the biasing and operation of the device. There are two types - NPN and PNP. NPN is the most common type. It has the higher speeds as electrons are the majority carriers and these have a greater mobility than holes. When run in common emitter configurations, the NPN circuits will use a positive rail voltage and negative common line; PNP transistors require a negative rail and a positive common voltage
V_{CEO}	Collector emitter voltage with base open circuit
V_{CBO}	Collector base voltage with the emitter open circuit
V_{EBO}	Emitter base voltage with collector open circuit
V_{CEsat}	Collector emitter saturation voltage
V_{BEsat}	Base emitter saturation voltage
I_C	Collector current
I_{CM}	Peak collector current
I_{BM}	Peak base current
I_{CBO}	Collector base cut-off current
I_{EBO}	Emitter base cut-off current
h_{FE}	Forward current gain
C_c	Collector capacitance
C_e	Emitter capacitance
F_t	Frequency Transition is the frequency where common emitter current gain falls to unity, i.e. the gain bandwidth product for the transistor. It is, typically measured in MHz. The operating frequency of the transistor should normally be well below the transition frequency
P_{TOT}	Total power dissipation - this is normally for an ambient temperature of 25C. It is the

	maximum value of power that can safely be dissipated for that transistor with its stated package
T_J	Junction temperature - care must be taken to ensure that this figure is not exceeded otherwise the device could be damaged or long-term reliability affected. Dissipation / temperature curves are often provided to facilitate calculations
T_{amb}	Ambient temperature
T_{stg}	Storage temperature is the temperature range over which the device may be stored. Outside this range, damage may occur to the materials used in the device
	based on table from this web page

A transistor's function is to control one current flow (controlled) using another one (controlling), which provides two operation modes: switch (on/off) the controlled current and controlling its strength based on the controlling current. Therefore, there are three regions of operation:

1. Cut-off

In this mode, the transistor is fully OFF. $I_C = 0$, $I_B = 0$, $V_{BE} < 0.7V$ and V_{CE} equals voltage applied between collector and emitter. Always make sure V_{CE} does not exceed this, as specified in the transistor specifications. In the OFF mode, the transistor is an "open switch" as no current flows.

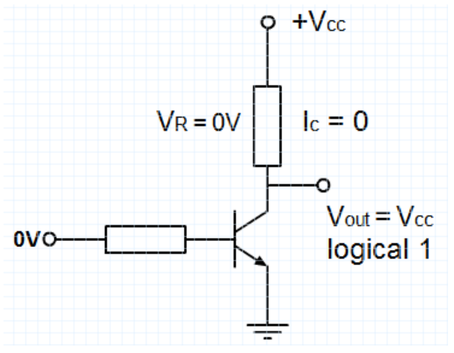
2. Active

In the ACTIVE mode the transistor operates as an amplifier ($I_C = h_{FE} \times I_B$). Changes in I_B reflect on the I_C. We aren't particularly interested in this mode at the moment, but if you want to read about practical applications regarding active mode, please read this material.

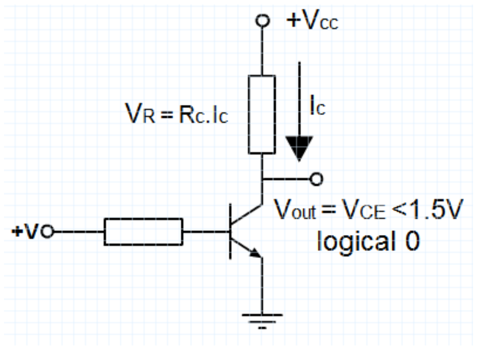
3. Saturation

In this mode, the transistor is fully ON. I_B is at the maximum allowed base current, so we can get the maximum possible I_C, $V_{BE} > 0.7V$ and V_{CE} is almost 0V. Always make sure I_B and I_C don't exceed the maximum current that the transistor can handle. If necessary, use a current limiting resistor to lower the current. In this mode, the transistor is a "closed switch", as current flows freely through it.

In the typical usage of the transistor as a switch, both "switch modes" (cut-off and saturation) are in use and usually have some load retained in the transistor's collector node. In this way, the transistor switches on/off the current flow releasing the load. Sometimes the load itself is the device we want to control (as we're going to do in a minute), and sometimes the load is simply a resistor. The latter is a typical example in situations in which we want to use the transistor's switch output voltage to control another device. If we get the voltage between the collector and ground, then we can use it as a logical input for the next device in the chain. Bear in mind that when the transistor is in cut-off mode, V_{OUT} is *logical High* and in saturation mode V_{OUT} it is *logical Low*.

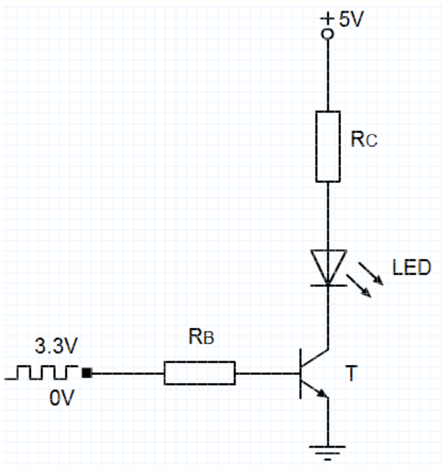


Logical High (1)



Logical Low (0)

Now back to our project. Look at the schema. This is a typical way to use an NPN transistor as a switch in saturation mode. Why's that? If no voltage is applied between the base and the emitter, the transistor stays "closed" and no current flows through the collector-emitter. As we apply voltage to the base, the transistor will "open" and the current will flow through the collector-emitter, powering the LED. In order to achieve this, we need to put the transistor T into saturation mode, which means $V_{BE} > 0.7V$. This voltage may vary between types of transistors, so firstly we need to choose T. I dug through my pile of parts and found BC 547, which is NPN BJT. A quick check of this transistor's parameters indicates that it will do the work. What we're looking for now is:



Parameter	What it is	Value	Actual
V_{CEO}	collector-emitter voltage	max 45V	5V (power supply)
V_{EBO}	emitter-base voltage	max 6V	3.3V (Netduino pin)
V_{CEsat}	CE saturation I_C	0.1V - 0.6V	0.1 (specifications)
I_C	collector current (DC)	max 100mA	20mA (LED current)
h_{FE}	current gain	min 110	10 (specifications)
P_{tot}	total power dissipation	500mW	≈ 2mW (calculated)

When the transistor is switched off, V_{CEO} will be 5V, that is, no current flows but the potential of the collector will be 5V). When the transistor "opens", then current will flow, and the voltage will drop around the resistor, LED and collector-emitter (CE). As the transistor is fully open, there will be no limit on the current that can flow through the CE, so we should be careful not to exceed the maximum one, i.e. 100mA, in order to prevent it from being damaged. In our case, this is far above the current we need to power on the LED, which is 20mA. A quick look at the specifications (figure 2

Saturation and on voltages) shows that when $I_C = 20mA$ then $V_{CEO} = V_{CEsat} \approx 0.1V$. The voltage is very low, so we'll ignore it in our further calculations, but to be on the safe side, we'll only use it this time to calculate how much power the transistor should dissipate: $P_T = V_{CE} \times I_C \approx 0.1V \times 20mA \approx 2mW$, which is far below its 500mW maximum. It's worth remembering that in cases in which a transistor is used as a switch (saturation mode), the voltage drop on the transistor is so small that we can ignore the maximum dissipated power and instead select a transistor by its max I_C. Also, when selecting a transistor for a switch, remember to pick one with a small V_{CEsat}. Transistors with $V_{CEsat} >$ 1V aren't suitable for this because of the high voltage drop, potential high power loss and temperature dissipation problems.

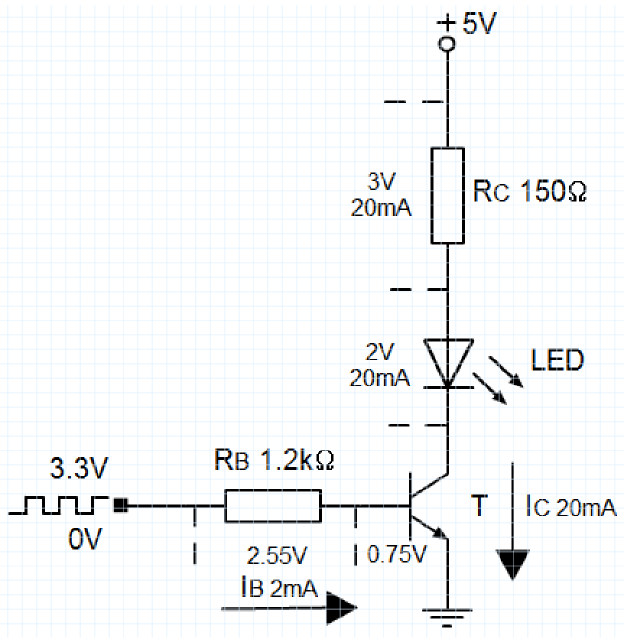
From the graph on figure 2 we can see that V_{CEsat} is such as previously mentioned when $I_C / I_B = 10$, meaning $h_{FE} = 10$. In our case we have min $h_{FE} = 110$ which will assure sufficient current through the base to guarantee saturation mode and max I_C. For our further calculations we'll use $h_{FE} = 10$. Next, we turn our attention to the base-emitter saturation voltage in the specifications V_{BEsat}, and we can see it's 0.7V when $I_C = 10mA$. Do you remember what we said about a transistor's modes of operation earlier? Saturation mode requires $V_{BE} > V_{BEsat}$. We can pick any voltage higher than V_{BEsat} and lower than the maximum allowed V_{EBO}, but let's do not overdo it, so we'll pick 0.75V. For base current we have $I_B = I_C / h_{fe} = 20mA / 10 = 2mA$. We can now calculate R_B which will be $(N_{out} – V_{BEsat}) / I_B = (3.3 – 0.75) / 2mA = 1275\Omega$. N_{out} is the Netduino output pin voltage for *Logical High* state.

Now, look here for standard resistor values. There is no 1275Ω under the E48 standard. It's common practice to replace one resistor value with the next closest upper or lower standard value. The closest standard values are 1.2kΩ and 1.3kΩ, but how to select the right one? A quick calculation shows that V_{BE} will be exactly 0.7V if we select 1.3kΩ and V_{BE} will be 0.9V if we select 1.2kΩ. Resistor values have some manufacturer tolerance anyway, so you can't be too precise, but instead you should consider the tolerance to be on the safe side. If you need to guarantee that you won't exceed the maximum current, then choose a higher value resistor so that the current will be lower. In our case, we have to guarantee saturation mode, which means a higher V_{BE}. Therefore, the lower value of 1.2kΩ is the logical choice. If we choose a 1.3kΩ resistor with +2% tolerance we can end up with $V_{BE} = 3.3 – (1326\Omega \times 2mA) = 0.65V$, which may prevent the transistor from achieving saturation mode. The same calculation for 1.2kΩ gives us $V_{BE} = 3.3 – (1224\Omega \times 2mA) = 0.85V$, well above our target of 0.75V. Just so that we have a clear conscience, we're going to check the opposite side of the tolerance -2%. $V_{BE} = 3.3 – (1176\Omega \times 2mA) = 0.95V$, which is well below the allowed maximum of 6V.

So, one thing to remember: *always keep tolerance in mind*. It makes calculations a little less precise and more complex. Later in this guide, we're going to see that the devices we build will accept tolerance, which solves our problems with precision.

Before going any further, we need to make sure the selected resistor can sustain the heat and won't melt down. That's easy: P_{RB} = (3.3 – 0.85) x 2mA ≈ 5mW. Note I used the lower value (0.85) of the tolerance range, in this way calculating the maximum potential dissipation of power. A glance here makes us feel good about ourselves, as we're well below the smallest standard power rating for resistors, which is 250mW.

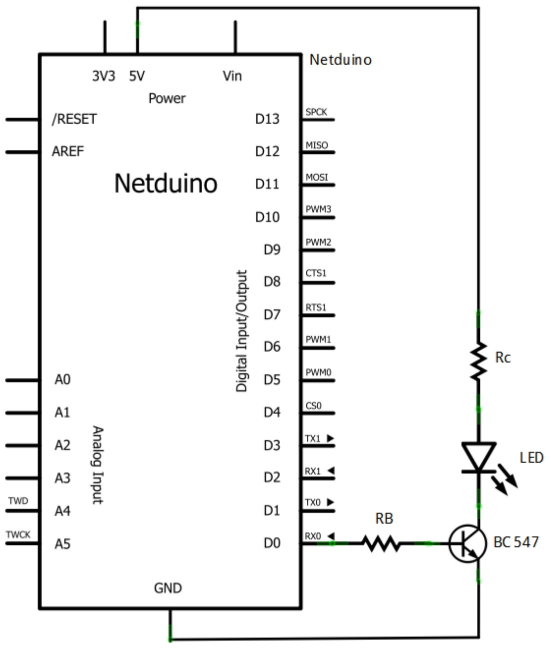
Finally, we can turn our sights to R_C (which we could have done in the beginning of the project, but in this particular case, order really doesn't matter too much). And so, $R_C = (V_{IN} – V_{LED}) / I_{LED}$ = (5V – 2V) / 20mA = 150Ω. That's a pretty standard resistor value. If you don't get this value, choose the nearest available but keep in mind what we mentioned before. Go for the nearest higher value to guarantee a current of below 20mA. On the power side we have P_{Rc} = (5V – 2V) x 20mA = 60mW, which is below 250mW. Just note that if it were higher the only thing we'd have to do would be to ensure we used a resistor from the higher power range, let's say 500mW, instead of 250mW. By the way, I didn't have 250mW resistors, so I used 500mW ones. This made my breadboard look uglier but who cares about beauty when we are after the beast☺. We can skip the tolerance calculations, as the difference will be insignificant.

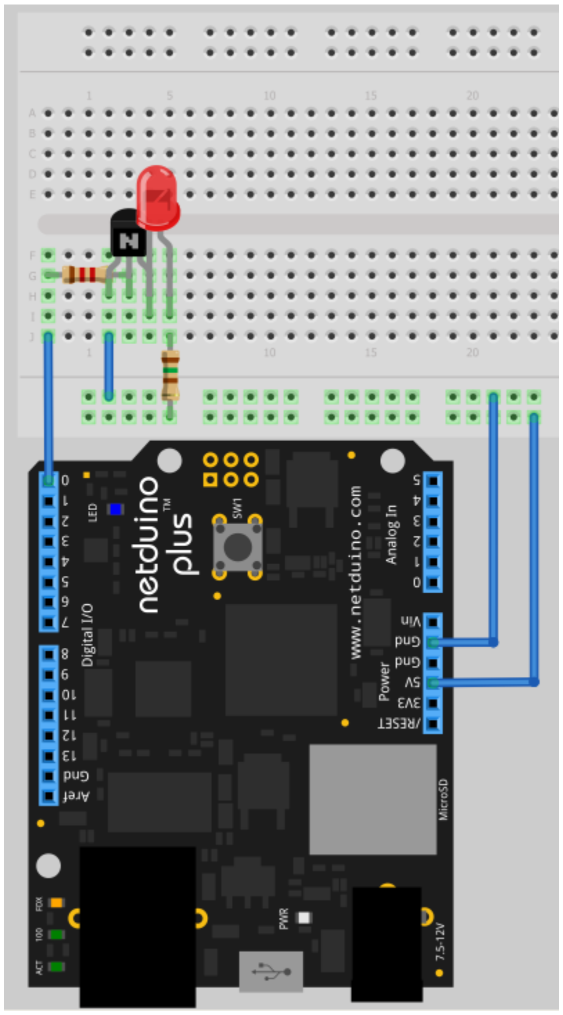


Let's have a look at the schema with the calculated values.

Wow, this was quite exhausting, and we're just at the beginning. Don't worry; things are going to be better from now on. For example, we don't have to re-do all those calculations for a yellow LED, as it seems the parameters are the same. There are tiny differences that won't cause any significant changes in the calculation, so we can safely assume that for this schema red and yellow LEDs are interchangeable.

What's left now is some DIY to get all the parts and then assemble them on the breadboard.





Exercises:

1. Recalculate the schematics if we were to change the LED to a high intensity blue LED (I_{Fmax} = 30mA and V_F = 4.5V), V_{CC} is 12v and using BC251 transistor.
2. Even if it's not good practice, we can power a LED directly from Netduino. What value of resistor would we have to use to do that? Will the LED achieve its full brightness? Can you tell why?

Part II (programming)

Time for some programming.

In the first place, we need a development environment. There are tons of excellent step-by-step materials of how to install it so just grab one from the Netduino web site and follow it. From now on, I'll assume you've done it and the environment is installed and ready for battle. So, let's write some code. Create a new project appropriate for your Netduino. I have N+ but this won't make any difference as we won't use anything specific to the type of Netduino, such as MicroSD or Network.

We're going to write a few simple examples, gradually increasing in complexity, which will help you get used to some basics of the hardware, .NET Micro Framework and Netduino SDK.

Boring LED flashing

(Source code for this example is in LED flash variant I project)

After creating a new Netduino project, use the following code inside Main() method:

```
public static void Main()
{
    var led = new OutputPort(Pins.GPIO_PIN_D0, false);

    while (true)
    {
        led.Write(!led.Read());
        Thread.Sleep(1000);
    }
}
```

Connect the breadboard and Netduino using Digital pin 0 as input for the schema's R_B, 5V pin and GND pin. In Visual Studio, right-click on the project file and select Properties. Select .NET Micro Framework tab and then select the correct transport to your Netduino board. Then build and deploy the application. If everything's OK, you should have the LED flashing. Note that during deployment Netduino reboots and at that point the LED lights up. The reason is that during this period Netduino's I/O pins are defaulted to input with its built-in pull-up resistor enabled, which causes the pins to have a "high" voltage level (3.3V), hence the transistor switches the LED on. We'll deal with this problem a bit later, but for now, let me explain the code.

It's pretty straight-forward to understand. We define a local variable led as OutputPort on pin D0 with a default status of ***false***. The application then goes into an infinite cycle that sets (Write) the OutputPort status to the alternative value of its current state (Read). Then the code puts the application's main thread into Sleep mode for 1s (1000ms).

What else should we know about the classes and methods we used?

1. OutputPort class

The MSDN documentation is quite sparse, but we're keen to know more about this class. From MSDN we can see that the class resides in the Microsoft.SPOT.Hardware namespace and inherits the Port class. Port class is used to manage the GPIO (General Purpose Input/Output) pins on the microcontroller. Each pin has a unique ID. They are declared as enumeration Cpu.Pin in Microframework (MF) and because MF is intended for generic usage the names in the enumeration are generic. In comparison, the SecretLabs SDK provides us with the SecretLabs.NETMF.Hardware.NetduinoPlus namespace containing the class Pins, in which we can find Netduino-specific pin names. Some of these are in addition to the CPU.Pins, like ONBOARD_LED and ONBOARD_SW1, which makes developing Netduino-specific applications a bit easier. Note that on Netduino, analog I/O pins (GPIO_PIN_A0 ... GPIO_PIN_A5) can also be used as digital I/O where needed.

OutputPort class has two overloaded constructors. There are a few cases where we'll need the constructor to set Resistor and glitch filter. For most cases, we'll only need to set the initial state of the port, so you'll find yourself mostly using the OutputPort (Pin, Boolean) constructor. This constructor requires two parameters, the Pin and its initial state - ***true*** to set the port to *High* (3.3V) or ***false*** for *Low* (0V). The actual voltage may vary slightly, but it's close enough to those values so we can accept them as the Netduino specifications state. Note that for other microcontrollers (like Arduino for example) those values, especially *High state*, could be different, so make sure you check this in the documentation.

In general, any input voltage less than 1.3V is interpreted as *logical Low*, and any input voltage over 1.5V, up to 5.5V, is interpreted as *logical High*. 1.5V is a threshold voltage and voltages over 5.5V will damage your controller. As an exception, in some cases *Low* and *High* can be recognised or accepted in the opposite of this case: *Low* as higher voltage and *High* as lower voltage. For now, we won't concern ourselves with those cases, but just bear in mind to always check the specifications of the devices you work with, as Logic threshold voltage level values vary between the different types of digital devices you use. A detailed diagram can be found here. The page also contains a link to Low Voltage Logic devices as well as a link to Bus Interface Logic devices. Even if we don't need all this information at the moment, it's good to have a general overview on the subject. Playing with microcontrollers implies that you will deal with Glue Logic devices quite a lot, so you also need to become familiar with how to select them. Good introductory tutorials on the subject can be found here and here.

For Netduino these voltage level values come from the microcontroller's specifications (page 610), and for *Low* (V_{IL}) they are from -0.3V to 0.8V and for *High* (V_{IH}) from 3.0V to 3.6V. To simplify calculations we assume that **Low = 0V** and **High = 3.3V**.

We also accept that ***logical Low state*** in electronics is represented as ***false*** in programming and ***logical High state*** is represented as ***true***.

Now that we have the output port constructed and initialised, we can use the Write(bool state) method to set and bool the Read() method to read its state. One thing to mention here is that Write() is part of the OutputPort class itself, whereas Read() is inherited from Port class. This doesn't directly relate to our example, but it would make sense once we come to best programming practices, where some principles insist we put our methods in the right place.

Here's a summary table of all we've learnt so far:

OutputPort(Pins.GPIO_Pin_D0, X)	Write(X)	X = Read()	Level	Voltage
False	False	false	Low	0V
True	True	true	High	3.3V

2. Thread class

If you've developed applications before, you probably know enough already to skip this section. If you haven't, you can read an introduction to threads here. We're more interested in how MF implements this and how we can use it. Here's an excerpt from the MF wiki: "Threads on the .NET Micro Framework is a subset of the implementation on the full .NET framework. It allows you to run multiple pieces of code (semi-)simultaneously. Running multiple threads is called multithreading. A good example of a multithreading environment is OS's like Windows and Linux, which allow you to run multiple programs at the same time. The basics of multithreading in C# are easy but multithreading can cause some extra programming pitfalls". We're going to skip multithreading for now, but we'll come back to it later. What we need to know and understand at this point is that our application will execute a single thread, so in this way every line or branch of our code will consume all available execution time from the MCU (if you're not sure what this is check here ☺). In other words, the MCU will use all its resources to execute our code as fast as possible. Where this may be fine in many cases, it's not such a good idea in some other cases. Our little application algorithm is a case that can be considered to be fine in both situations, depending on what we want to achieve. Still not convinced?

In this particular stage, our code implementation of the algorithm is coming to the loop

```
while (true)
{
        led.Write(!led.Read());
        Thread.Sleep(1000);
}
```

What will happen? Any guesses? Well, Thread.Sleep(1000) causes the thread that executes the cycle to sleep for one second. But this particular thread is the one that runs the whole application (main thread), which means MCU will be "busy sleeping" for a second. This may still be fine for simple and quick examples, but it's not very practical when it comes to power efficiency or application responsiveness. Netduino excels in both, saving power as well as being responsive, but it's up to us to allow it to do so. We can content ourselves for now with knowing that there is a better way… and carry on.

Before leaving this subject, have a look at the MSDN documentation about System.Threading.Thread. Wow! Quite a big class, isn't it ☺? Don't worry! You can learn it one-step at a time.

Flash the LED until I press the button

(Source code for this example is in LED flash variant II project)

In this project, we're going to relax a bit, so we won't be jumping immediately into any "heavy and confusing" electronic design and calculations. Well, that's not entirely true, because we have to encounter some electronics now and then, but no calculations, I promise. We're going to do some more programming instead and try to learn more about .NET Microframework.

The easiest way to do that is to make use of Netduino's onboard switch. For this example, we don't have to know too much about how it's actually working from an electronics point of view, but if you want to do that, the best place of course is the Netduino board schematic (this is a link to N+ schematics but it's the same in terms of the switch). Just look for "Push button" in the top left corner of the schema.

From a programming point of view, accessing the switch is almost as simple as accessing the OutputPort. Let's have a look at some new code inside the Main() method:

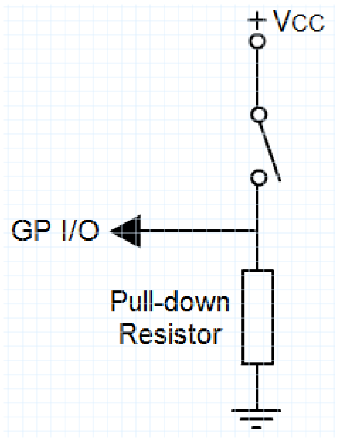
```
public static void Main()
{
        var led = new OutputPort(Pins.GPIO_PIN_D0, false);
        var button = new InputPort(Pins.ONBOARD_SW1, true, Port.ResistorMode.Disabled);

        while (!button.Read())
        {
                led.Write(led.Read());
                Thread.Sleep(1000);
        }
}
```

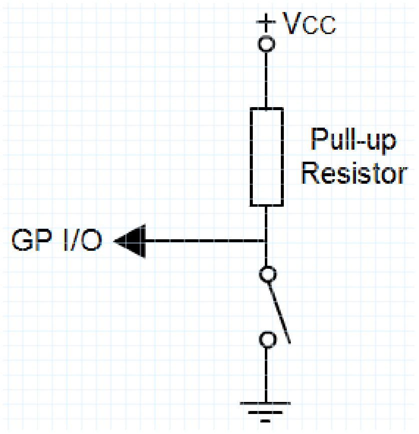
From here, InputPort class "represents an instance of an input port that can be used to read the value of a GPIO pin". First, we declare the button variable and assign to it a new object of type InputPort that will represent the ONBOARD_SW1. Nothing too complicated so far. Now we reach a part that requires a bit more explanation.

Nothing in the real world is ideal, which is why we should consider things like power supply fluctuations, EMI and so on, which can cause a "noise" that interferes with and might cause strange or unexpected results from a GPIO pin. In our case, using just a switch could cause the so-called "bouncing" effect. You can read more details on this effect here and here (search for "Contact bounce"). This is where the second parameter comes into place – the glitch filter. Setting the glitch filter to ***true*** will cause all potential noise to be filtered when the MCU reads the port. Setting ***false***, of course, will not consider the noise. The glitch filter works through imposing a small delay of 7.68ms (by default) before detecting the logical state. Where did 7.68ms come from? Well, that's a long story and later it could well become a subject for discussion. For now, all we need to know is that it has something to do with the way real time intervals are represented through the frequency of the generator that the MCU uses. In our case, the default value is not what we need, so we can change it in the following manner:

```
Cpu.GlitchFilterTime = new TimeSpan(0, 0, 0, 0, 10); // 10ms
```

Next, our attention comes to the ResistorMode parameter. Documentation about this is again quite sparse, which is why we're going to explore it a bit more in order to gain a better understanding of what it's all about. MCU recognises *logical High* (3.3V) as ***true*** and *logical Low* (0V) as ***false*** on its input ports, but as the inputs themselves are open collectors there's also a third state – floating. This third state is considered as an *undefined logic state*, neither ***true*** nor ***false***. We don't have a way of representing this state as Boolean, so somehow we have to eliminate it. That's where so-called Pull-up and Pull-down resistors come into play. Have a look at the following schemas and try to figure out yourself how it works.





In the case of a Pull-down resistor, when the switch is open and the GPIO is in an undefined state, this resistor ties down the GPIO to 0V, so any noise and any floating voltage will be grounded, giving us a clear ***false*** state. When the switch is closed, V_{CC} (3.3V for Netduino) can be detected at the GPIO and we have a clear true ***state***. V_{CC} vary between different schematics, but in Netduino's case it has to be 3.3V (Netduino GPIO are 5V-tolerant, but do avoid this where possible). We already discussed this subject in detail here. In the case of a Pull-up resistor, things are reversed: we have a clear ***true*** state when the switch is open and a clear ***false*** state when it is closed. Note also that without using a resistor, when the switch is pressed, the current will flow directly to the ground and no voltage will be present on the GPIO port, so we can say that the resistor is the reason we can detect the switch state.

Now we have to remember that Netduino's MCU GPIOs have *built- in* Pull-up resistors that we can set programmatically using ResistorMode.PullUp, but also remember that Netduino's MCU *does not have a built-in* Pull-down resistor. It means we can use the *built-in* one where needed to set a clear ***true*** state as default, but we have to add an *external* Pull-down resistor when we need to set a clear ***false*** state. If we don't include an external Pull-down resistor, then the ResistorMode.PullDown option is completely useless.

The third value in the ResistorMode enumeration, Disabled, is an interesting one. People usually misunderstand this option and think that it will disable the port, which is not true. This mode means that neither a Pull-up nor a Pull-down resistor will be wired to the port. For Netduino's MCU we won't use the built-in Pull-up resistor, but will instead use an external Pull-up resistor to set a ***true*** state, as well as an external Pull-down resistor to set ***false***.

For those of you who didn't look into the Netduino board schematics, I should add that the on-board switch is already wired up with a Pull-up resistor. This means we'll have ***true*** when the switch is open and ***false*** when the switch is pressed. This is a potentially confusing situation, which is why the exit of this schema is wired to the /SWITCH1 pin, which inverts (explaining the "/" in front of the pin name) the logic and makes it easier to follow. Note that it's the firmware that actually inverts the result from these readings. The prewired Pull-up resistor (note that it's prewired as external to the MCU) also makes using the MCU's built-in GPIO Pull-up resistor redundant, and in this way ResistorMode.Disabled and ResistorMode.PullUp will produce the same readings. It's widely accepted practice that ResistorMode.Disabled should be used when reading the value of the onboard switch and the result is:

On board switch state	GPIO Read()
not pressed	False
pressed down	true

You can observe those values during debugging using Debug.Print() method in the following manner:

```
while (!button.Read())
{
        Debug.Print(button.Read().ToString());

        led.Write(led.Read());
        Thread.Sleep(1000);
}

Debug.Print(button.Read().ToString());
Thread.Sleep(5000);
```

Values will appear inside VS's Output window. Sleep(5000) is used simply for convenience and will give us some time in which to see the last read value before the debugging session is closed. Without this, we can still see what the values were by navigating to VS's Output window. We'll learn more about Debug.Print() method later.

Last thing to pay attention to in this project is the loop. You may ask, "What's so important about this loop? The one we use already is fine". Well, that's absolutely true. The reason behind talking about the loop is that this loop evaluates the condition for looping *before* any code inside the loop is executed. In most cases, this is exactly we want to do, but in some we may want to execute the code inside, *at least ones before* the condition's evaluation has been performed. So instead of the ***while*** loop in the previous example, we can use a ***do … while*** loop:

```
do
{
        led.Write(!led.Read());
        Thread.Sleep(1000);
}
while (!button.Read());
```

Using this loop will cause the LED state to be set at least once. You can test the following: using our first code for a ***while*** loop, reboot the microcontroller but press the switch on after the onboard LED turns off (meaning the reboot has finished) and keep it pressed. The LED does not switch on at all, as the code inside the loop is never executed. Now, try the same with a ***do … while*** loop.

Become ECO friendly

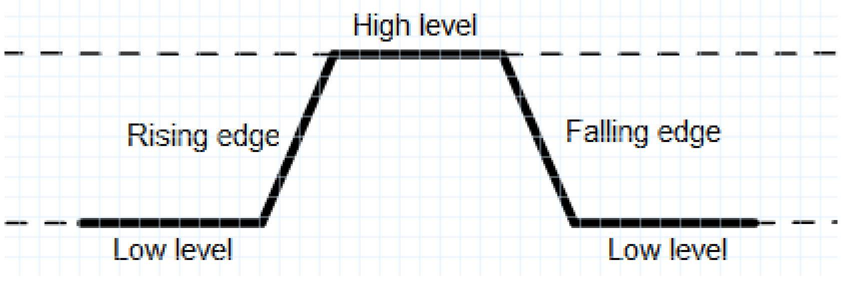
(Source code for this example is in LED flash variant III project)

As mentioned before, the approach we used for flashing the LED while the on-board switch is pressed is good, but it costs energy. The MCU is running on full power all the time and if we power it from a battery for example, this will exhaust it in no time. What can we do? We should think of a better way of flashing the LED and checking the switch.

Timer is a good way to flash the LED efficiently. Timer constructor uses a TimerCallback delegate method that is executed each time a specific time interval passes. This way we don't have to worry about measuring this interval ourselves, the hardware will do that for us, and we still have control over exactly what we want to do when this interval passes. Using Timer and TimerCallback delegate is very easy, as you'll see when we come back to the LED flashing timer code implementation, but before that, let's solve the efficiency problem for the switch.

You may have thought that a timer is a potential way to check the button status too. Well, yes, it's possible, but it's not very efficient. To react swiftly, we have to set a very small interval for the timer, which will negate our effort to save energy. No, we have to think about a better way and fortunately, Microframework has the solution for us – InterruptPort.

Using InterruptPort we can give the MCU time to recover and to go into power saving mode. Instead of madly checking the switch state, the MCU processor now delegates this task to the MCU I/O hardware, instructing it to be notified (interrupted) if a particular event happens. In this case the event will be pressing on the switch button. However, exactly what happens on a hardware level? To understand that we need to look at the following diagram:



As you can see, the input signal (event) has four distinctive states. Each one of them can be used as an interrupting event that causes the hardware to wake up (trigger) the MCU processor in order to execute specific code. "*Low*" and "*High*" level statuses can be considered as stable levels, whereas "*Rising*" and "*Falling*" edges occur where the status is in the process of shifting from one stable level to another.

Interrupts are quite a flexible and powerful mechanism for creating responsive, real-time hardware devices that are controlled by software. External events cause some sensor changes that trigger the interrupt. Meanwhile the MCU is free to execute any other tasks until an interrupt occurs. When the MCU is "interrupted", it suspends the execution of the current task and turns its attention to the interrupt event. In this way, it looks like it reacts to the event. When the event is processed, the MCU turns from the interrupt to resume execution of the postponed task.

Let's turn from the signal states and have a look at how they are represented in Microframework, which is done through Port.InterruptMode enumeration. Apart from the states that we already listed, this enumeration contains two more: InterruptNone, which simply means disabling interrupt events, and InterruptEdgeBoth, which can be used to react when any change of status occurs (meaning when the signal is in process of transferring between states in either direction).

Time for us to go back to the actual implementation of all that we've said up to this point. There are a few different approaches to specify callback (event) delegates. We're going to present the most common and the most "modern" way. Below is the most common implementation approach:

```
public class Program
{
        private static TimeSpan ledFlashInterval = new TimeSpan(0, 0, 0, 1);
        private static TimeSpan initialDelay = new TimeSpan(0, 0, 0, 3);

        private static Thread mainThread;

        private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, false);
        private static InterruptPort button = new InterruptPort(Pins.ONBOARD_SW1, true,
Port.ResistorMode.Disabled, Port.InterruptMode.InterruptEdgeLevelHigh);

        public static void Main()
        {
                mainThread = Thread.CurrentThread;

                // Attach button listener
                button.OnInterrupt += new NativeEventHandler(ButtonEvent);

                // Set and start LED flashing timer
                var ledFlashTimer = new Timer(
                        new TimerCallback(LedFlash), null, initialDelay, ledFlashInterval);

                // Main thread goes to sleep
                mainThread.Suspend();
```

```
            // Stop the LED flashing timer
            ledFlashTimer.Dispose();

            led.Write(false);
        }

        private static void LedFlash(object data)
        {
            led.Write(!led.Read());
        }

        private static void ButtonEvent(uint port, uint state, DateTime time)
        {
            mainThread.Resume();
        }
}
```

First we declare two TimeSpan variables that represent LED flash interval and initial delay (to delay initial flash and could be set to 0). We also declare mainThread, a variable to contain the main application execution thread, so we can access it inside the child thread. We already know how OutputPort and InterruptPort work. In Main(), we assign an actual value to the mainThread, then we create and assign a button (which been declared as InterruptPort), a new NativeEventHandler, and pass ButtonEvent as a callback method, which will be executed when a button is pressed. ButtonEvent() itself simply uses mainThread to access the main thread and Resume() it (in the documentation you should heed the CAUTION advice in the Remarks block. Despite this advice, we can still use this pair of methods because the main thread will never have been locked). Ignore the code regarding ledFlashTimer for now. What will happen then? The main thread will be Suspended until we resume it by pressing a button. The benefit is that in being suspended, the main thread allows the MCU to enter low power mode. Great – this is exactly what we wanted.

Now back to the declaration of ledFlashTimer, which we declare as Timer with TimerCallback LedFlash(). In this code, we don't care about passing any data to the method so we pass **null** and the two TimeSpans. The LedFlash contains the (by now) well-known led.Write(!led.Read()) which alternates LED ON/OFF. Later, when the button is pressed and the main thread resumes, we Dispose of the ledFlashTimer object, in this way stopping the timer and freeing all associated resources.

Now, let's see a more "modern" implementation approach, using lambda expressions:

(Source code for this example is in LED flash variant IV project)

```
public class Program
{
        private static TimeSpan ledFlashInterval = new TimeSpan(0, 0, 0, 1);
        private static TimeSpan initialDelay = new TimeSpan(0, 0, 0, 3);

        private static Thread mainThread;

        private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, false);
        private static InterruptPort button = new InterruptPort(Pins.ONBOARD_SW1, true,
Port.ResistorMode.Disabled, Port.InterruptMode.InterruptEdgeLevelHigh);

        public static void Main()
        {
                mainThread = Thread.CurrentThread;

                // Attach button listener
                button.OnInterrupt += new NativeEventHandler(
                        (uint port, uint state, DateTime time) =>
                        {
                                mainThread.Resume();
                        });

                // Set and start LED flashing timer
                var ledFlashTimer = new Timer(
                                new TimerCallback(
                                        (object data) =>
                                        {
                                                led.Write(!led.Read());
                                        }),
                                null, initialDelay, ledFlashInterval);

                // Main thread goes to sleep
                mainThread.Suspend();

                // Stop the LED flashing timer
                ledFlashTimer.Dispose();

                led.Write(false);
        }
}
```

Initially, I wanted to write a detailed explanation about this way of writing code, but I thought it was better to leave this task to you. The reason for this is simple: the code speaks for itself. Technically, it's exactly the same thing we've done in the previous code implementation, but this time instead of declaring separate delegate methods, the body code is directly inside lambda expressions, and I strongly suggest that you become more familiar with them. In the examples provided here you will find how this second method can be done in a slightly different manner, moving the ledFlashTimer declaration up to the fields and thus making the Main() method even shorter.

The method of implementation you choose to use is entirely up to you but the result remains the same, so enjoy programming in your own way and your own style.

There are a few more things you should be aware of about InterruptPort. If a trigger is set to be at a stable level ("High" or "Low"), an event will then only be triggered once, unless we specifically perform "interrupt clearing". When the expected event occurs, the level is stable and therefore it seems logical that we have no more interest in it after an action is performed. In fact, this is exactly what the code example is doing. It detects first that a button is pressed and triggers the appropriate event, after which it ceases to care about further events of such a type. As part of processing the event, it resumes the main thread, which in turn goes further and ends with program execution. Even though this looks logical and practical, there are situations when we actually want to keep reacting to the event. To do so, we have to “clear” the interrupt port and allow it to receive and trigger future events of this type. We can achieve it by calling ClearInterrupt(), which is one of the InterruptPort class members, in the following manner:

```
private static void ButtonEvent(uint port, uint state, DateTime time)
{
        // Do things before enable interrupt again
        ...

        button.ClearInterrupt();

        // Do things after enable interrupt again
        ...
}
```

or

```
button.OnInterrupt += new NativeEventHandler(
        (uint port, uint state, DateTime time) =>
        {
                // Do things before enable interrupt again
                ...

                button.ClearInterrupt();

                // Do things after enable interrupt again
                ...
        });
```

For status-transferring events, such as Rising and Falling edge, there’s no need to use clear interrupt. These events are transitional and logically we expect another state to follow them. Therefore, Microframework does not require any actions from us and will keep accepting events.

Interrupt could be temporarily disabled with DisableInterrupt() and re-enabled with EnableInterrupt(). Other methods to disable interrupt is setting its status to set port interrupt Port.InterruptMode.InterruptNone (button.Interrupt = Port.InterruptMode.InterruptNone). This method is very useful in case we want to change the type of the triggering event:

```
        ...
        // Disable interrupts
        button.InterruptPort = Port.InterruptMode.InterruptLowNone;
        // Set further interrupts to be triggered on Low level
        button.InterruptPort = Port.InterruptMode.InterruptLevelLow;
        // At that point interrupts are allowed
        ...
        // Disable interrupts again
        button.InterruptPort = Port.InterruptMode.InterruptLowNone;
        // Set further interrupts to be triggered on High level
        button.InterruptPort = Port.InterruptMode.InterruptLevelHigh;
        // At that point interrupts are allowed
        ...
```

And, of course, use Dispose() to force resources associated with the port to be cleared.

In this chapter, we made significant progress in our journey into the Microframework world. You should already have noticed that despite its name, this framework is quite powerful and we can find all sorts of jewels and gems inside. I'd advise you to keep learning it yourself as well as to continue reading this tutorial.

Exercises:

1. With the glitch filter set to true, try briefly pressing the switch button. Notice that it doesn't interrupt the application until you press the switch a bit longer. Now set the glitch filter to false and try again.
2. Repeat the same experiment with setting glitch filter true/false, but also change the event to be one of the transferring levels (Rising or Falling edge).

Better Multithreading

(Source code for this example is in LED flash variant V project)

Despite learning how to do multithreading, we won't stop in our attempts to improve what we've achieved already. Look back at the code and try to find lines that look a bit awkward. Yes there are☺. Look at mainThread.Suspend() and mainThread.Resume(). They do the job but read strangely somehow; we put the main thread to sleep and at the same time, a child thread is still running and even later resumes its parent. Obviously, it's working, but it seems a bit strange. We won't go too deeply into the additional problems this approach can cause, but I will mention that it merits additional caution where two or more threads share the same resource, like the same variable for example. In cases like that, we have to implement a locking mechanism to prevent one thread changing the variable in such a way that will result in the other obtaining invalid data. Sounds bad, so this is why we have to find a better way to do this. Because of those potential problems, Suspend() and Resume() have been deprecated in favour of a better approach.

.NET Microframework provides a way of synchronising threads using events. Using this approach, one thread can wait until another thread finishes its work, and then and only then will it continue. For this to work, Microframework provides us with the specialised class WaitHandle. This is a base class for two additional classes: ManualResetEvent and AutoResetEvent. For the WaitHandle class, it clearly states "Encapsulates operating system specific objects that wait for exclusive access to shared resources", so the class is really all about managing shared resources. Even though we don't have any shared resources in our code, it's still better to use this specialised class to manage multithreading because it provides better flexibility and extensibility, which can be useful when we need to expand the code to something more complex.

There's only one main difference between the two additional classes ManualResetEvent and AutoResetEvet – ManualResetEvent requires manually resetting the status of the event after processing it. This might be useful if you need additional control in when to allow further thread synchronisation, but for now, we'll prefer to use AutoResetEvent.

An excerpt from the MSDN documentation says, "AutoResetEvent allows threads to communicate with each other by signalling " We'll use it to improve communication between the thread detecting on-board switch status and the main thread.

Reading this code now makes more sense. After creating the button interrupt and creating the flash timer, the main thread waits (listens) for one of them to signal that it's finished, so that the main thread can continue its work. In this particular case, only the button will signal, but, as a further example, we could include a counter to count the number of times the LED flashes and only after that to signal the main thread. Can you implement this code yourself?

Now compare your solution with following two variants (note the difference in where ledFlashTimer is instantiated).

First variant:

```
public class Program
{
	private static TimeSpan ledFlashInterval = new TimeSpan(0, 0, 0, 1);
	private static TimeSpan initialDelay = new TimeSpan(0, 0, 0, 3);

	private static Thread mainThread;

	private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, false);
	private static InterruptPort button = new InterruptPort(Pins.ONBOARD_SW1,
true, Port.ResistorMode.Disabled, Port.InterruptMode.InterruptEdgeLevelHigh);

	private static AutoResetEvent syncEvent = new AutoResetEvent(false);

	public static void Main()
	{
		mainThread = Thread.CurrentThread;

		// Attach button listener
		button.OnInterrupt += new NativeEventHandler(
		(uint port, uint state, DateTime time) =>
			{
				syncEvent.Set();
			});

		// Set and start LED flashing timer
```

```
            var ledFlashTimer = new Timer(
                        new TimerCallback(
                              (object status) =>
                              {
                                    led.Write(!led.Read());
                              }),
                        null, initialDelay, ledFlashInterval);

            // Main thread waits for signal
            syncEvent.WaitOne();

            // Stop the LED flashing timer
            ledFlashTimer.Dispose();

          led.Write(false);
      }
}
```

Second variant:

```
public class Program
{
      private static TimeSpan ledFlashInterval = new TimeSpan(0, 0, 0, 1);
      private static TimeSpan initialDelay = new TimeSpan(0, 0, 0, 3);

      // Set and start LED flashing timer
      private static Timer ledFlashTimer = new Timer(
                  new TimerCallback(
                        (object status) =>
                        {
                              led.Write(!led.Read());
                        }),
                  null, initialDelay, ledFlashInterval);

      private static Thread mainThread;

      private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, false);
      private static InterruptPort button = new InterruptPort(Pins.ONBOARD_SW1,
true, Port.ResistorMode.Disabled, Port.InterruptMode.InterruptEdgeLevelHigh);

      private static AutoResetEvent syncEvent = new AutoResetEvent(false);

      public static void Main()
      {
            mainThread = Thread.CurrentThread;

            // Attach button listener
            button.OnInterrupt += new NativeEventHandler(
                  (uint port, uint state, DateTime time) =>
                  {
                        syncEvent.Set();
                  });

            // Main thread waits for signal
            syncEvent.WaitOne();

            // Stop the LED flashing timer
            ledFlashTimer.Dispose();
```

```
                led.Write(false);
        }
}
```

Solving the problem with LED becoming ON when Netduino boots

At this point, you probably noticed that the LED comes ON when you switch on or restart Netduino. This is because in fact MCU ports go into High state during hardware initialisation or software reset. In most situations, this would be fine, but imagine if we need this LED or other, controlled schematics switched OFF until we explicitly turn it ON. Well, we'll solve this problem now.

Since there are many different approaches we can choose from, I decided to show only three of them in detail and a few others as a general overview. I'm sure as you become more experienced as electronics designers, you'll find more and better solutions. After all, we have to improve things around us, don't we?

In general, the idea we're going to work around is simple: prevent the schematics getting any power, until the Netduino booting process is finished and the application executes. In fact, the final implementation will give even further control – to power ON/OFF the schematics programmatically.

Relay approach

The first thing that comes to mind when talking about controlling a power ON/OFF is, of course, Relay. Relays have a special magical aura that causes everyone to use them for almost everything – just joking☺. Still, relays provide an easy and quick solution for lots of things, even though they're not very elegant electronic devices. They're also quite reliable and have a long record of service. Anyway, enough attempts to win you over to the relay cause and let's keep going.

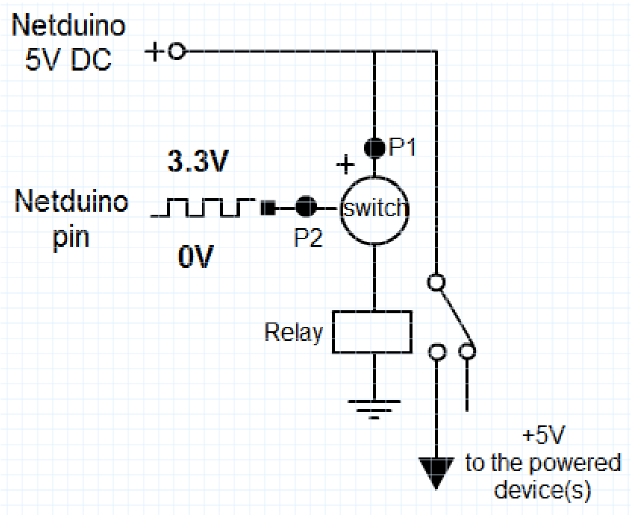
Some considerations we have to keep in mind in planning the following schema:

1. Where we will get the power supply?
    a. We'll use Netduino's 5V DC but should be careful not to exceed the maximum specified 800mA. This current is more than sufficient for what we'll do in our next few projects, so for now we'll stay with this option, but just be aware of other possibilities:
    b. We can use an external power source. If it's 5V, this will save us needing to recalculate the parts. If we need higher than that, we can reapply our knowledge of how to do it, so this should be an easy-peasy job.

2. Netduino digital pins are 3.3V (input and output) but we'll use a 5V source. Normally we have to be extra careful at all times not to give the pins 5V. At this point, it might not be clear to you what this means, but in the process of designing it you will understand this point. It's good for us that Netduino pins are 5V tolerant, which will keep us from burning it even if we make a mistake, but don't count on it – change to an external supply with 6V or more volts and if you're not careful, you'll end up with a nice piece of plastic pizza ☺.
3. At all times, also keep in mind that there is an internal Pull-Up resistor, which can potentially mess with what you are doing. At the very least, doing a quick double check of the final schematics operation against this resistor is necessary. A few times, I found myself puzzled why schematics that I calculated so precisely were not working as expected. Measuring the voltages across the scheme showed strange values and puzzled me even more, until I realised that I forgot to count the internal Pull-up resistor. It sounds crazy, but it can have you hitting your head against the wall for days☺.

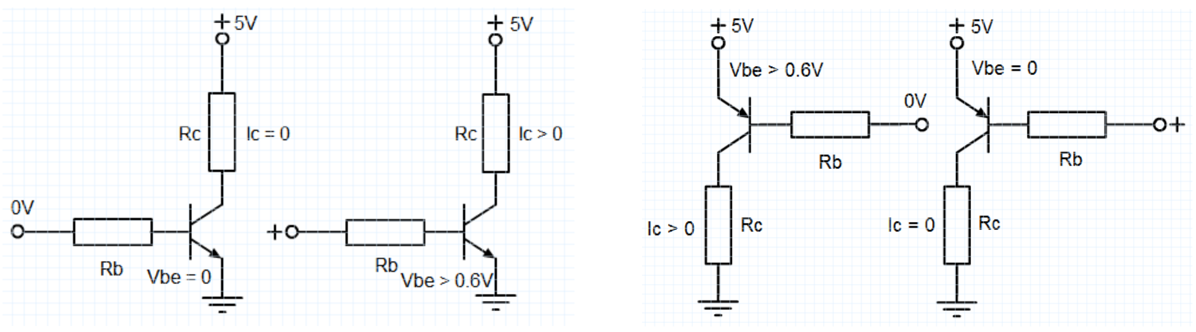
In trying to achieve our goal, we'll go through a more detailed process of designing and calculating the schematics; in this way those of you who don't have much experience in electronics design will get a better impression of how this goes. With more experience in this process, you won't even consider following Stage 1 design but again, this comes with experience.

Stage 1 scheme



On the right is a basic diagram of what we want to do in principle. The electronic switch will control the relay, through which we supply power to the rest of the devices. What's interesting about the switch itself is the difference in potential between P1 and P2. P1 is 5V where P2 varies between 0 and 3.3V, which gives us a 5 to 1.7V difference. This is important as that's the key towards biasing the switch correctly, but it's also a difference in potential that could damage Netduino's pin (remember point 2 of the consideration list). Imagine we used a 6V instead of a 5V power supply. Then also imagine we forgot that the pins are tolerant up to 5V and somehow (because of wrong or bad schematics) we apply that 6V to the pin! I hope by now you have the picture☺.

The next point to decide on is what kind of switch we'll use. We already know one kind – the NPN transistor switch – but this one goes ON where there is any voltage higher than 0.6V applied to its base and, unfortunately for us, this is exactly what happens immediately after Netduino is switched ON. Between this point and the code execution, where we're able to lower the pin's state, the interval could be quite long, so the NPN transistor switch is not quite the one we need. However, there is another transistor switch – the PNP transistor switch – which we can use. In principle, it works in the same way as the NPN transistor switch – any voltage higher than 0.6V (0.7V when saturation mode is required) between the base and emitter will open the transistor, but the main difference is that this time the current flows from the emitter to base. Let me explain the difference:



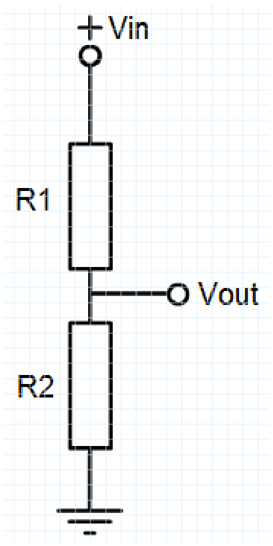
When using the NPN, base current flows from base to emitter and for this to happen, we have to apply enough voltage to the base so it will create >0.6V potential between B and E. When using a PNP transistor, the situation is similar, but this time the emitter should be more positive than the base, in order to create current that flows from emitter towards base. Note also that this time R_C is between the collector and ground. If MCU pin output voltage is same as Vcc voltage – 5 volts, then the following logic operation results are valid from a load (Rc) point of view:

NPN transistor switch		
Input	Ic	Output
false – 0 (Vbe = 0V)	0	false
true – 1 (Vbe > 0.6V)	>0	true

PNP transistor switch		
Input	Ic	Output
false – 0 (Vbe > 0.6V)	>0	true
true – 1 (Vbe = 0V)	0	false

Now back to our case, where we have a 3.3V pin that's less positive than the 5V supply and the difference is 1.7V. That's far higher than the saturation voltage we need and it looks like the transistor will be open all the time. So how then can a PNP switch solve our problems? Well, if we just could lower this 1.7V below 0.6V, things would probably be fine, but this only solves cases where the pin is in a high state. What will happen in situations where the pin goes to a low state? That's the beauty of using a PNP transistor. In cases where the pin is in a high state (3.3V), the difference in potential will be 1.7V, but where the pin goes to a low state (0V), then the difference increases to 5V. Having the difference increase is actually the right direction in which we need it to go. Remember, we want to switch the transistor ON when the pin goes low and to switch it OFF when the pin goes high. So up to this point, everything looks promising and it seems that the only problem left is to find a way of lowering the potential to less than 0.6V when the difference is 1.7V, and at the same time to be more than 0.6V when the difference is 5V. Let's familiarise ourselves with the Voltage divider.

The Voltage divider makes use of the voltage drops around two resistors to create a split point. We can use both of those dropped voltages as a new voltage source for the next part of the schematics, but typically V_{out} is considered to be between the split point and ground. Below are the basic equations you'll need to calculate the voltage divider, depending on what you know and what you need to calculate.



Main equation is: $\frac{Vout}{Vin} = \frac{R2}{R1+R2}$

Derivate:

$Vout = Vin\frac{R2}{R1+R2}$ $\quad$ $Vin = Vout\frac{R1+R2}{R2}$

$R1 = \frac{Vin.R2}{Vout} - R2$ $\quad$ $R2 = \frac{R1}{\left(\frac{Vin}{Vout} - 1\right)}$

When using a voltage divider, always keep in mind that there are two currents flowing through R_1 (one that flows through both resistors and one that flows through R_1 and the load connected to V_{out}) and one through R_2. The current through the divider is $I_D = V_{IN} / (R_1 + R_2)$ from which you can find the power that the resistors have to dissipate.

We'll use a voltage divider to lower the 1.7V and 5V to more appropriate values that we can apply to the transistor's base.

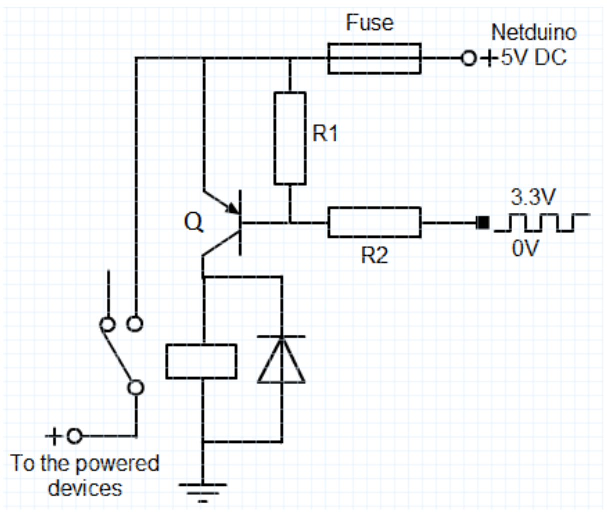
Now we are properly armed to put all we know about the PNP transistor switch, the Netduino pins and the voltage divider into place so that they work together. Before reading on I suggest you spend a few minutes and try to draw the scheme yourself. Use your imagination and knowledge; even if it takes a few hours, you can spend that time practising some ideas, which isn't wasted time but learning time. The more you practise such exercises, the better you'll be when you need to create schematics yourself. After all, there will be a time when there is no ready example to follow and you have to do it alone.

Before we go any further, there are two other things to mention:

1. The influence of Pull-Up resistor

Fortunately, it doesn't mess up our scheme, but let's have a look at this. During boot time the Pull-Up resistor makes the pin go up to 3.3V, but this only affects the difference between P1 and P2. Why doesn't it affect us in any other way? Because we will use this difference to keep the transistor switch closed. At the time we will be able to control the pin state we will configure the port and in doing so we will eliminate the Pull-Up resistor.

2. The last thing to note is the fact that setting the pin to a lower state means grounding it, so the current flows from emitter, base, then goes to the ground through the pin.



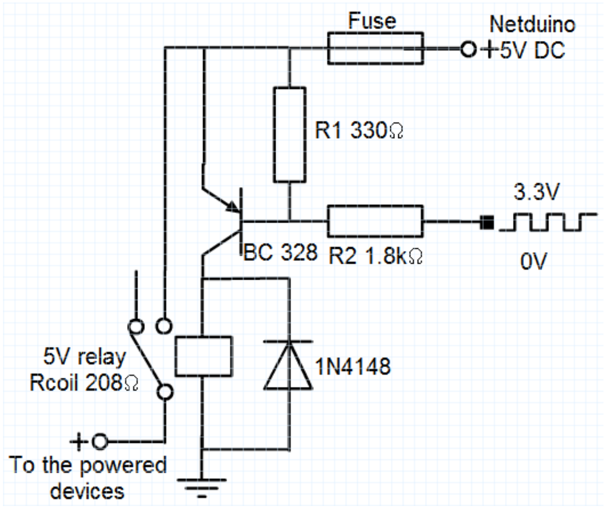
Stage 2 scheme

As you can see from the schematics, R_1 and R_2 form the voltage divider that we've spoken about. To calculate the resistor values we need to consider the I_B, hence we first need to choose a transistor. I've chosen BC 328, and you can see the transistor's datasheet here. Following the chain of requirements, we come to the point where we require the value for I_C. This value, naturally, comes from the value of the Relay's coil resistance. Therefore, we choose a relay, which has to be a 5V relay. I used a 5V Omron G6M relay with R_{coil} = 208Ω, from which we can calculate that I_C = 5V/208Ω = 24mA. Going back to the transistor's base current, from all we know about using a transistor as a switch and the BC 328 datasheet, we find that I_B = I_C / 10 = 2.4mA. Knowing the base current, we can turn our attention to R_2. To calculate its value ,we should consider that the I_B (2.4mA) will flow when Netduino's pin, attached to the base, will be 0V and that V_{BE} is 0.75V, so voltage applied to R_2 is V_{out} = Vcc - V_{BE} = 5V – 0.75V = 4.25V, which gives us R_2 = V_{out}/I_B = 4.25V/2.4mA = 1770Ω. We'll therefore use a 1.8kΩ (250mW) resistor. Now, using our knowledge about the voltage divider, we can calculate that $R1 = \frac{Vin.R2}{Vout} - R2 = \frac{5V.1.8k\Omega}{4.25V}$ - $1.8k\Omega = 317\Omega$. The nearest higher standard value is 330Ω. Do you remember what we've said about tolerance? I'll leave it to you to calculate why it's better to choose a higher value instead 300Ω.

That was easy, wasn't it☺? Did we miss anything? Yep, we had to check what V_{BE} would be when the pin is 3.3V. Voltage difference will then be 5V – 3.3V = 1.7V which gives us a voltage value of $Vout = Vin\frac{R2}{R1+R2} = 1.7V\frac{1.8k\Omega}{300\Omega+1.8k\Omega} = 1.44V$. This is excellent news because it means V_{BE} = 1.7V

- 1.44V = 0.26V and this voltage certainly isn't enough to open the transistor, and so the switch will remain OFF.

You may wonder what a diode is doing in the schema. It's used to protect the transistor. When the current flows though the relay coil, it generates a magnetic field that makes the switch go from one contact to another. When the current is switched off, this magnetic field generates current inside the coil, thus producing a brief high voltage spike at the ends of the coil. At that moment, the diode comes into action and shortcuts the coil ends, draining the current. Without the diode, no current could flow and the coil's voltage spike can damage the transistor, as this voltage is in reverse polarity to the normal voltage. As you may notice, the diode is reverse-connected against the normal current flow, which is why it doesn't interfere with the rest of the schematics. Usually a signal diode is best suited to do this job as they react quicker to changes compared to rectifier diodes. I've chosen 1N4148.



At left are the final schematics including calculated values. For the fuse, I used a 750mA resettable fuse. While this isn't necessary, it's good to have in case you forget the 800mA limit and try to use more. If you power Netduino through the USB port, any attached devices will be able to consume no more than 500mA, so you may want to swap my 750mA fuse with a 500mA one.

In concluding the schematic part of this project, we must say that we learned quite a lot of new things, went through the entire process of designing the schematics: from the initial idea to potential solutions, identifying the right one and finally calculating the elements. Of course, this project is simple and is still missing a few parts of the bigger project's design, like circuit analysis etc. but I hope it serves its purpose of gently introducing the world of electronics to you.

Now how do you think we will control this, let's call it a "power switch"? Initially you may think OutputPort will be sufficient to control it, as in the following code:

```
// Switch ON power (remember low state makes voltage difference of 5V)
OutputPort powerSwitch = new OutputPort(Pins.GPIO_PIN_D13, false);

// Switch OFF power (remember high state makes voltage difference of 1.7V)
OutputPort powerSwitch = new OutputPort(Pins.GPIO_PIN_D13, true);
```

While it looks promising, there's a problem – we need to change the port direction. When the port is in a high state we want it to be OutputPort (no current flowing through the port), but when it's in a low state we need it to be InputPort, thus allowing current to flow through it (I_B current should flow through the pin and then to the ground). Well, it's easy to imagine that disposing of the port and re-declaring it as InputPort will probably solve the problem, but Microframework has a better solution for us.

The solution is a TristatePort. Even this class is very useful: it's a very peculiar one, because in its attempt to solve several problems at once, it becomes an elegant mass of confusion☺.

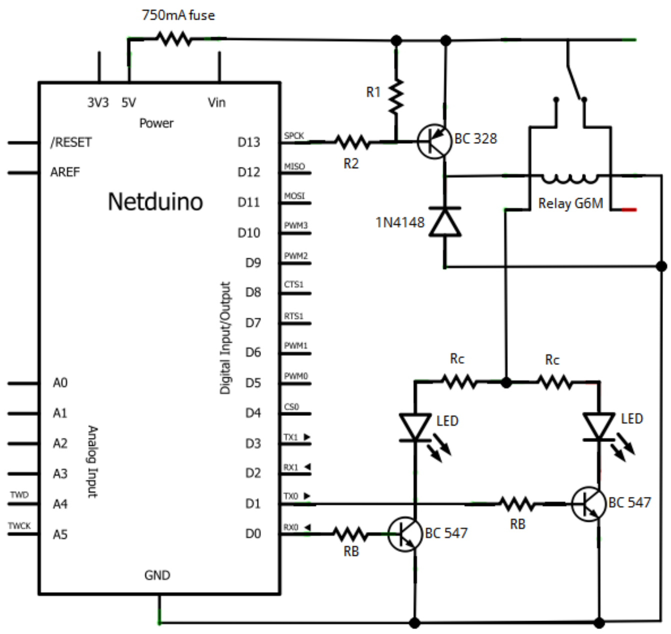
The first important thing to notice is that its name is highly misleading. It sounds like we can control the port to be in a High, Low or High impedance state. This isn't the case. It's all about how embedded devices implement Input/Output ports, the traditional implementation of which is a Tri-state register. These registers work as bi-directional ports. TristatePort represents this tri-state register, but it actually has only two operational modes – input and output, and at any point in time, the port can operate in only one of those modes. Its usefulness comes from the fact that, in contrast to InputPort/OutputPort, it can alternate between these modes on the fly without the need to redefine the port type.

TristatePort.Active property controls the operational mode. When set to ***true***, the port works as **output port (output mode)**, and when set to ***false*** the port then works as **input port (input mode)**. Note that you should never set Active property with the same value consecutively as this will throw an exception.

Next, more confusion comes from the TristatePort's constructor second parameter, named *initialState*. This parameter has nothing to do with the operational mode, but it specifies what will be the actual *output from the pin*, ***true*** or ***false***, when we put the port in **output mode (output port)**. In this mode we can change the output from the pin using Write() method. When TristatePort is in **input mode** we can Read() the state of the pin. Note that it makes no sense to read the port state when the port is in **output mode,** as well as no sense to write to it when it's in **input mode**.

After initialisation, by default, TristatePort operates as **input port** and the other two constructor parameters, *glitchFilter* and *ResistorMode*, are valid in this mode.

Just before illustrating in code how all of this works, let's look at a more practical example and combine the "power switch" schematics with two LED schematics.



I've added a second LED schema to illustrate the idea of using the "power switch" as a power source control to other various schematics that can be attached to Netduino. In the following code example, I haven't added any code to control this second LED, but have instead left this task for you as an exercise.

Before we go any further, you should now assemble all the schematics on the breadboard and come back for the code implementation when you are ready…

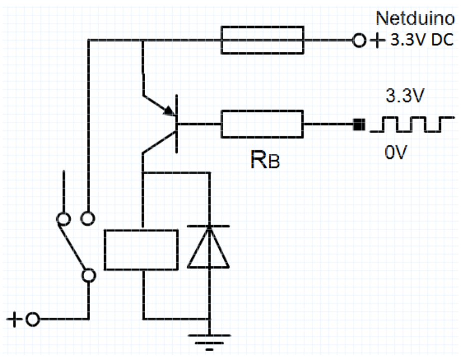
…It's time to write some code and bring the whole schematics to life. Using the code from the last LED-flashing example, we can add the "power switch" code:

(Source code for this example is in LED flash variant VI project)

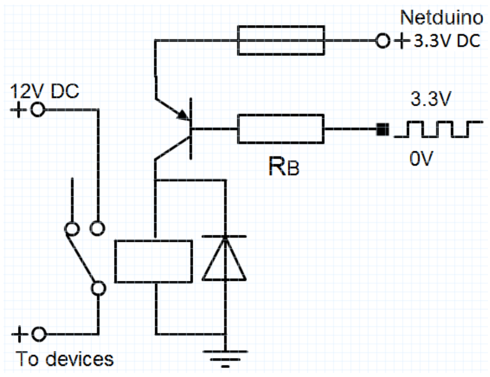
```
public class Program
{
        private static TimeSpan ledFlashInterval = new TimeSpan(0, 0, 0, 1);
        private static TimeSpan initialDelay = new TimeSpan(0, 0, 0, 3);

        // Set and start LED flashing timer
        private static Timer ledFlashTimer = new Timer(
                        new TimerCallback(
                                (object status) =>
                                {
                                        led.Write(!led.Read());
                                }),
                        null, initialDelay, ledFlashInterval);

        private static Thread mainThread;

        private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, false);
        private static InterruptPort button = new InterruptPort(Pins.ONBOARD_SW1, true,
Port.ResistorMode.Disabled, Port.InterruptMode.InterruptEdgeLevelHigh);

        private static AutoResetEvent syncEvent = new AutoResetEvent(false);

        // Define digital port 13 as tri-state port without PullUp resistor
        private static TristatePort powerSwitch = new TristatePort(Pins.GPIO_PIN_D13,
false, true, Port.ResistorMode.PullUp);

        public static void Main()
        {
                mainThread = Thread.CurrentThread;

                // Attach button listener
                button.OnInterrupt += new NativeEventHandler(
                        (uint port, uint state, DateTime time) =>
                        {
                                syncEvent.Set();
                        });

                // Switch "power" ON
                powerSwitch.Active = true;

                // Main thread waits for signal
                syncEvent.WaitOne();

                // Stop the LED flashing timer
                ledFlashTimer.Dispose();

                led.Write(false);

               // Switch "power" OFF
                powerSwitch.Active = false;
        }
}
```

We already know how the LED flash code works. For the power switch, we have to mention only one thing: when initialising the TristatePort we specify that, when port becomes Active = true (**output port**), its initial state is false (0V), thus switching the power switch on. We also keep the pull-up resistor in use, so when we set Active = false (**input port**) this resistor will make the pin's voltage 3.3V, thus causing the power switch to switch off the relay.

In retrospective, going back to our initial goal, we managed to use Netduino's 5V DC to power some schematics and at the same time managed to avoid the effect of powering those schematics at the boot/reset time. Some similar situations may occur in which we can use the same approach with small amendments, for example, we may use Netduino's 3.3V DC. In this case, we won't need the voltage divider, because there is no voltage difference (which we had in the case where we were using 5V DC). Then the schematics will simply be the same as an NPN switch and only one current limiting resistor connected between the base and pin, but still using the PNP property to switch ON when voltage goes 0V and stay OFF when voltage is 3.3V.

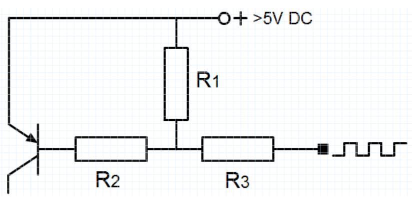


What we can do if we need a higher voltage to power controlled schematics? In this case, we can address the problem simply by connecting the Relay's controlled circuit to the higher voltage, as shown in the schema on the right. To control the relay, you can use a schema with 3.3V DC or the 5V DC one.



If you want to use the higher voltage to power the transistor switch, as we did in the 5V DC schematics, you should be very careful and avoid more than 5V reaching the Netduino’s 3.3V digital pin at any cost.

On the right, you can see example schematics that can solve this problem, but this time I’ll leave it to you, as an exercise, to find a way of calculating the values. A little tip: this schema makes use of Resistor ladder, but in a slightly different way. I will explain the Resistor ladder in more detail in a later section of the material. For now, if you wish, you may exercise and re-design the schematics to make full use of Resistor ladder.



Using SCR thyristor and our first steps into digital logic gates

In the previous approach, we used a relay to control the power and were quite successful, but having a relay on the board can make some people feel there is another and possibly better way to achieve the same result. Well, there is such a way and we will look into it. We’ll also use this opportunity to gain knowledge of a new electronic component and will introduce ourselves to the world of digital logic, using transistor NOT gate. Before going further with this material, I suggest you re-read the “Thyristor” and “Digital logic” sections in Introduction to electronics principles and components chapter of this tutorial.

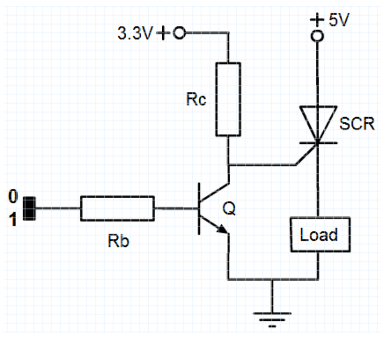
There will be no programming done in this approach, as we will reuse exactly the same code written in the previous approach. After all, we are going to change only the controlled schematics and not the application logic.

The component that we’ll replace the relay with is a Thyristor. For the purposes of this project, from the variety of thyristor types we’ll use SCR C106D Logic level thyristor.

C106D’s general description states: “Passivated, sensitive gate thyristor in a plastic envelope, intended for use in general purpose switching and phase control applications. This device is intended to interface directly with microcontrollers, logic integrated circuits and other low power gate trigger circuits.” Its working parameters are within our desired limits: we need to pass 750mA through it and this is well below the 2.5A value stated in the specifications. We’ll apply 5 volts to it and this too, is far below the 400V that C106D can sustain. Apart of those values, we’re most interested in the values listed in the “STATIC CHARACTERISTICS” table. I’ll reference them in the following text at the places where those parameters come into use.

Now, let's think about ways to control the SCR. C106D can directly interface with the MCU, according to its description, but we'll use a transistor NOT logic gate to control it instead. We'll do this to be on the safe side (not interfacing C106D directly with the MCU, even the specs state it's safe to do so) and for learning purposes.

Now, the schematic we'll use looks like this:

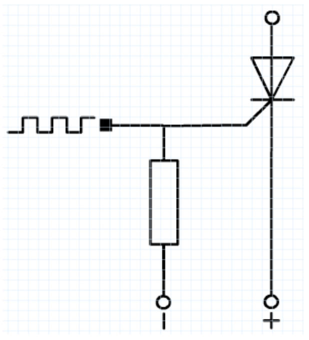


Can you imagine how this schematic works? Note the two different (3.3 and 5V) voltage sources. I've done this intentionally to make things a bit more difficult, thus you can learn a bit more (some additional considerations are in place in this situation). Later, as an exercise, you may re-calculate whole schematics for two 5V sources.

Let's start at the beginning. When we switch ON MCU, the ports are exposing a logical true (1) in form of 3.3V. This causes the transistor to fully open. Because of this, there are a few things happening. First is that the transistor is working as a shortcut and sinks all the current flowing from 3.3V through Rc. The second thing is very important for us. It makes the current (if any) at SCR gate flow in the opposite direction – "from" the gate, instead "to" the gate. In other words, it makes the SCR reverse-biased. Those two things cause the SCR to stay OFF (it takes some current to flow "to" the gate to switch ON the SCR). Keep this in mind. It'll come in very useful in the last scenario we'll describe.

Now when we set the pin to false (0), 0V applies to the transistor's base thus closing it. This stops the sinking effect the transistor had in the previous state. Now any load connected to the SCR's cathode will cause current to flow from 3.3V through Rc, then to the SCR gate, then through the cathode, load and then to the ground. This opens the SCR and the main current from our 5V source will flow through the SCR and the load. In this mode, we'll normally have to achieve at least the SCR's holding current (I_H) to keep it open, but in our case there will be a constant gate current, which will hold it open regardless of the actual current value. This constant current through the gate should also make us careful about the value of Rc. It has to be high enough in order to limit the SRC's gate current as well as to a level that won't burn it.

At last, we come to the most interesting part – switching the SCR OFF. This is slightly complicated to explain. Usually, to switch OFF the thyristor, we have to use methods mentioned in the "Thyristor" section. Here, this rather unusual design makes use of an additional property of the sensitive gate SCRs have.



We can increase the SCR's holding current using a method known as reverse biasing. We can achieve reverse biasing by adding a resistor between the gate and cathode or ground (in our case ground) and applying reverse voltage as shown on the schematics on the right. The holding current depends reciprocally on the value of the resistor – a smaller resistor increases the holding current more than a bigger resistor.

In our initial design, there's no such resistor as a discrete element, but instead the tiny resistance of the transistor's collector-emitter junction takes that role. Because this resistance is very low, we can increase the holding current quite significantly. On the other hand, this means that there won't be sufficient holding current to keep the SCR open when we remove the triggering signal from the gate. We have to be very careful not to burn the transistor, as we'll be exposing it, even just for a very short period, to current that could potentially be higher than that it can sustain. Similarly, we should be aware that the 3.3V voltage source will be exposed to a small reverse voltage. In our case, this is insignificant and can be ignored because the Rc value (calculated later) is quite high, thus making the reverse current tiny. This tiny current will be applied to the voltage source, so we have to make sure it won't burn the voltage source. We'll return to these problems later and see how, where it's required, we can use a small resistor attached between the gate and transistor's collector to eliminate this danger. On the other hand, we shouldn't add a very high value resistor, because this will lower the holding current and we may lose the ability to switch OFF. Unfortunately, there will be cases when we'll have to sacrifice this ability, but we'll still be able to switch ON the load **after Netduino (re)boots** (*remember this is the main goal*).

Now, think where the reverse voltage will come from. At the moment we open the transistor, the gate will become far less positive, it's practically directly grounded, than the cathode which is grounded too, but the load's resistance is far higher than the collector-emitter junction's (remember current flows from more positive to less positive potentials as well as finds the path with smallest resistance).

Combining all these factors, we can conclude that the SCR is switching OFF because of the momentum reverse biasing which causes current, flowing through anode-cathode, to fall below the increased holding current value.

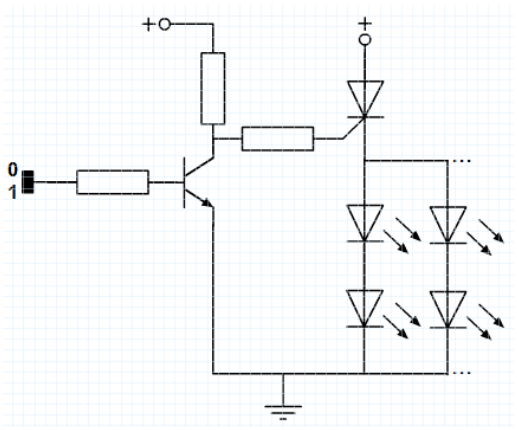
Here I'd suggest you try to perform the required calculations yourself. This will be good exercise for you. Don't worry if you don't succeed. The attempt is important and if you have even a few correct assumptions and calculations, this is a very positive sign that you will do much better in future.

It's time for calculations, some of which you already know how to perform. The transistor switch part of the schematics is functionally the same as the one in project one, so we can reuse the same parts: BC547 and R_B = 1.2KΩ. On the other hand, you can redo the calculations as an exercise or pick other parts and see what values you come up with.

I already mentioned that for this project we'd use C106D SCR. Here you can find its datasheet. You can replace it with MCR22-6 or any similarly sensitive gate SCR (if there's a high audience demand about the subject of finding replacement parts then I'll include a chapter on the subject, so feel free to message me through the Netduino forum or leave a comment on the tutorial's post or wiki). First, we have to check the max forward current and max voltage that the thyristor can sustain 2.5A/400V (all following data is for C106D). Next is checking the gate parameters: peak forward and reverse gate voltages (measured between the gate and cathode) are 5V, which is fine, as our schematics will operate inside the 5V range. The gate trigger voltage should be anything over 0.1V and not more than 1.5V. The most important parameter is the gate trigger current, which for this SCR should be between 15µA and 200µA. Now the only value we need to calculate is Rc. I've chosen a value of 100µA for the gate current (a nice round one) and calculated that Rc = 3.3V/100µA = 33KΩ. Why did I use 3.3V? Well simply because switching the gate will happen at the moment when the transistor is OFF, thus all current will flow from the 3.3V source, through RC and then to the gate. Now to complete our knowledge of this schematic, let's see how much current from 3.3V will be sinking when the transistor is open. You're right, it's the same amount. Two more resistances that we can ignore are the resistance of the gate-cathode junction in the first case and resistance of collector-emitter junction in the second. They're tiny, so their influence is insignificant. I also ignored the voltage drop across the gate-cathode. Is that being imprecise? Let's recalculate with this drop in mind: Rc = (3.3V – 0.7V)/ 100µA = 26KΩ. However, would this matter? What would happen if we keep Rc = 33KΩ? In this case, what would change would be the current, so I_G = (3.3v-0.7V)/33KΩ = 78µA. This is more than enough to switch ON the SCR. So I'll keep Rc = 33KΩ. Why am I doing this? Because if you choose to be more precise and pick Rc = 26KΩ, it really won't affect the functionality or stability of the schematics. The difference would be that you would sink more current in choosing 26KΩ than in using 33KΩ. Why consume more power without good reason? Returning to the resistor, what's left is to check the power that the resistor will dissipate: 33KΩ x $(100µA)^2$ = 330µW. Here I beg you: please don't touch this resistor, your finger will freeze immediately☺.

That's it! No more calculations are required. You can now replace the Relay schematics from the first approach with this one. No changes in application are required so it's ready to go.

After you test this new schematic, just for fun and some programming practice, attach the LED and the 150Ω resistor used in Project one, as load. Then write a wee application that switches ON/OFF the thyristor repeatedly. Here you can see this in action with both C106D and MCR22-6. Now you have another way to blink a LED. ☺



For those of you that are not yet convinced about the effect of reverse biasing SCR and that the resistor value is reciprocal to the increase of holding current, I'd suggest the following experiment: get 20 LEDs, connect them in pairs with 2 LEDs in series (as shown on right), then use those pairs as load. This will increase the current flowing through the load and SCR significantly. Now connect a 150Ω resistor between the gate and the collector and Rc. You will lose the ability to switch OFF. Now lower this resistor to 56Ω and the switching OFF ability will come back.

Using NAND gate (Negated AND) and current "sourcing"

Now we'll turn our attention to more sophisticated methods of solving the problem with the LED turning on when the MCU boots. First, we'll look at how to eliminate this effect directly for the schematic from Part I. I recommend you have a quick overview of the material in this section before continuing further. Later, when we'll be armed with solid knowledge how to solve this problem on small scale "per pin", we'll jump back to solving this problem on bigger scale, as we did with relay and SCR. Don't despair – all we'll learn is really interesting and includes basics and principles that are "must know" for every beginner.

In previous approaches, we used a more radical solution: we controlled the power of the subsequent schematic. Now let's try to find ways to control the input of the schematic instead. From what we know is happening on MCU boot, the pins are in high-impedance state, which the transistor switch recognises as logical high and switches the LED on. It's obvious that we have to find a way either to prevent the transistor recognising logical high or to reverse output from the MCU pin. Many proposed solutions use a pull-down resistor, which lowers the voltage and thus the transistor no longer recognises logical high. Unfortunately for us, things aren't so simple as they seem and there are a few problems with this approach. One problem is that it is very difficult to calculate and find out the correct pull-down resistor value. This is because MCU already has an integrated pull-up resistor which itself has some tolerance difference, thus you can find yourself in a situation where the same pull-down resistor is working absolutely fine on one pin but is behaving strangely with another pin. Eventually you'll make things work somehow, but will you feel comfortable with this try/fail approach? Another problem, which I think is even more significant, is that you will sink current especially, and with every additional pull-down resistor this will drive more current from the power supply, which will flow through your MCU, potentially causing it to dissipate more heat. If you're OK with this, then stop reading here!

Still here? Good, I'm glad!

It seems that what's left to us is to use the second approach: reverse the output logic. We can't do that on MCU or Netduino board level, so we'll have to come up with some schematic to do the job for us. The good news is that there are many ways to do that. We'll have a glance at a few of them, which are very basic and as simple as possible. As usual, we'll do this for learning purposes. I hope that after reading the next few pages you'll be well-armed for the future with more knowledge and more importantly with the understanding that your desired outcome can be achieved in various ways. This is important for beginners, as I've seen how easily people can lose their enthusiasm after their first failure in trying something they thought would work. Please, don't lose your enthusiasm, but instead look for another way to solve the problem!

Let me present to you the most complicated solution first. After that, we'll look into easier and simpler ways.

In this approach, we'll compare the pin's logical output to something with a constant "high" level: the pin's output compared to the 3.3V or 5V power supply. Why have I mentioned the two voltages? Because there's no practical difference from logic's point of view – both are high level.

This is also the place to mention that in the following examples we'll look first on the solution using IC chip and then on its BJT equivalent. We also will use an old fashion TTL IC. I know there are many arguments about this subject but TTLs are simple to handle from a beginner's point of view. Their biggest advantage is that you won't have to worry about static electricity damaging your ICs. Later, when you become more experienced, you should switch to CMOS but definitely do start with TTL.

Now the question is how we'll compare those logical levels? This is where a NAND logic gate will come to our rescue. One of the gate's inputs will be connected to the 3.3V/5V power supply line and the other one will be connected to the pin, the output of which we want to invert. It's so simple

- NAND will compare the logic levels of the two inputs and will produce the desired logic result on its output.

Let's see the logic in the following table view:

NAND input pin 1 (level is set by PSU's 3.3V/5V)	**NAND input pin 2** (level is set by MCU pin)	**NAND output pin**
High (***true***)	High (***true***)	Low (***false***)
High (***true***)	Low (***false***)	High (***true***)

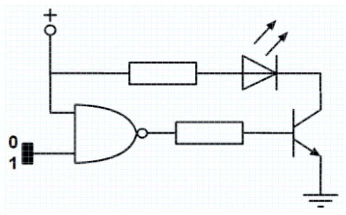
Note that the NAND output is opposite to the NAND input coming from the MCU pin. That's exactly what we want as a result. When MCU boots or restarts, the pin will return true which will register as false on the NAND's output. However, did you spot the logical problem we have now? We have to send false to the pin to switch on the LED. This is a bit confusing, yeah. Don't worry, we'll solve this "problem" a bit later when we look into the program. There's a nice trick to make the logic "straight".

Now back to the transistor switch schematic. We'll have to do a tiny adjustment to the schematic and in addition we'll have to take care of a few specifics related to the IC that we'll use: 74LS00 (SN74LS00N to be more precise about the exact IC I used, so download its datasheet form here).

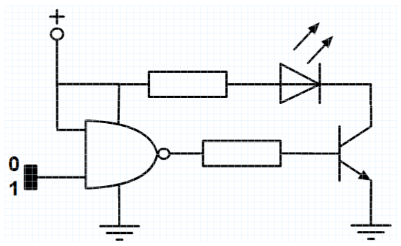
7400 series is a family of TTL IC that contains various types of devices like logic gates, flip-flops and even ALU. Later you'll probably want to switch to 4000 series, which is similar, but a CMOS set of devices. The specific characteristic of the ICs inside each series have quite similar properties like, for example, input sensitivity or output logic levels. They may differ in some IC specific functions or parameters, but in general, if you know some of the common parameters then you already know quite a lot for the whole series.

Here, I suggest you try to draw the schematic yourself. Also, think about how the transistor switch schematic I mentioned earlier could be adjusted. All the information you'll need to come to the final result is included inside the IC datasheet. Read carefully, don't rush, take your time, make few attempts. You don't have to finalise your design in 5 minutes, better that you spend a few days. Take some notes on what you consider is important and what isn't.

I'm expecting you to come up with something very similar to the schematic on the right. You can think about this schematic as a functional or principal one. Its goal is to express the solution in a



very generic way. However, why is it only a generic one?



Well, there are few elements missing from it. For example, NAND IC has no power source connected to it. OK, let's add the connections.

However, we still missed something. Do you have a clue?

What's missing is a power decoupling capacitor. The theory and practice of power decoupling is very interesting in itself and I'll spend a little time explaining it.

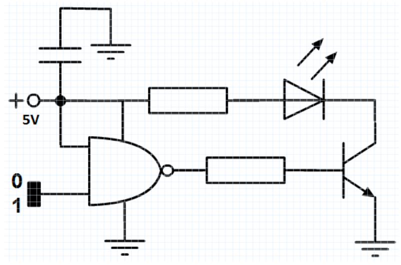
In the first place, why do we need a decoupling capacitor? To be more precise this is "transient load decoupling" and we have to consider it where there will be a process of fast switching. Ideally, the switching should happen instantaneously. In the real world, however, logic gates tend to do sudden switching, thus, from load's point of view, there's a moment of middle voltage. To eliminate this effect, we need to provide enough current to the load during the moments when IC switches from one state to another. That's where a capacitor can help, as it acts as an additional power supply and keeps the current constant, but having a capacitor in place isn't as simple as it sounds.

Because the process of power decoupling includes using a capacitor, most beginners think that it's all about the capacitor. In fact, we're adding a LC adjacent to the IC. You may ask where that inductance comes from, when there is only a capacitor in place? Well, in fact a few "hidden" inductances should be considered too, such as the capacitor's inductance, the wires or interconnecting traces and the IC itself (which mostly depends on the type of IC packaging). For example, some typical values are 1-2nH for capacitor, 5-20nH for wires and board traces and 4-15nH for the IC. This gives some 10 to nearly 40nH inductance! Therefore, we should be well aware that we're not just adding a capacitor but a whole LC circuit. What does it matter? An LC circuit has oscillation capability, which gives the LC its own resonant frequency, which has to be taken into consideration when things start moving too fast.

When frequencies are less than 50MHz best practice shows that a 0.1µF ceramic capacitor, placed as close as possible to the $+V_{CC}$ pin of the IC, will do the job in almost any case. Note that at the power supply end, we need a similar capacitor to ensure that the power supply is as steady as possible. The good news is that because we'll use the Netduino's regulated power supply, we won't need an additional capacitor - it already has one.

For frequencies between 50MHz and 500MHz, a single capacitor per IC isn't enough. A whole series of capacitors has to be distributed across the board. Things are even more complicated for speeds over 500MHz. We won't concern ourselves with these methods, as for now we'll stay well below 50MHz.

With the addition of a decoupling capacitor, the schematics will look like the one on the right.



Is that it, though?

Up until now, we've only used one of the NAND gates of the SN74LS00N IC. The chip itself contains four NAND gates. In addition, if you read the small print (always read the small print) in the documentation, then you'll probably notice that we need to tie all unused inputs of the IC either to the V_{CC} or to the ground.

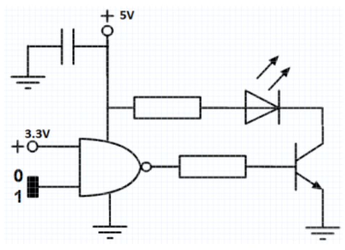
Some logic family chips may require the pins to be connected to ground or V_{CC} with an additional resistor. The resistor serves two goals: to reduce noise susceptibility and to protect the pin by limiting the current from high variations of voltage source that could damage the input. The exact value of this resistor depends on a few factors, but it usually is in the range of 1kΩ and 10kΩ. There's no need to use a resistor with 74(x)S00 (note letter S) series because the inputs are internally protected by Shotky diodes.

How can we decide where to connect input pins to ground or V_{CC}? The easiest thing to do is to apply the following logic: it's better to set a configuration of the inputs that produces a low output state. For example, with NAND it's better to connect the input pins to V_{CC}, thus leaving the outputs in low. If it was an AND gate, then it's better to ground the inputs. Of course, some cases could be more complicated that the one we have here, but this strategy works well in most situations.

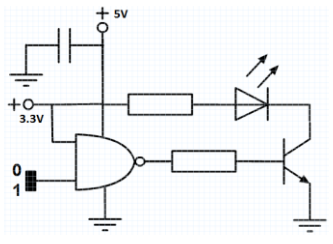
What should we do with the output pins? Simple: leave them unconnected.

So, remember to tie unused input pins to V_{CC} when you come to assemble the final schematics on the breadboard!

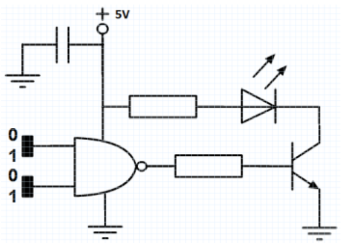
Now, before we go for some calculations, I want to mention that the these schematics use the +5V power source as logic level for comparison, as the power source for IC, and finally as the power source for the LED. All that simplifies the schematics (remember we already discussed why it doesn't matter if we give a 3.3V or 5V as input to the TTL IC). If you're concerned by this factor or by any other one, then you may go for designs like the following:



It compares MCU pin output against 3.3V



It uses 3.3V to compare to MCU pin and to power the LED



It makes use of an additional MCU pin for comparison

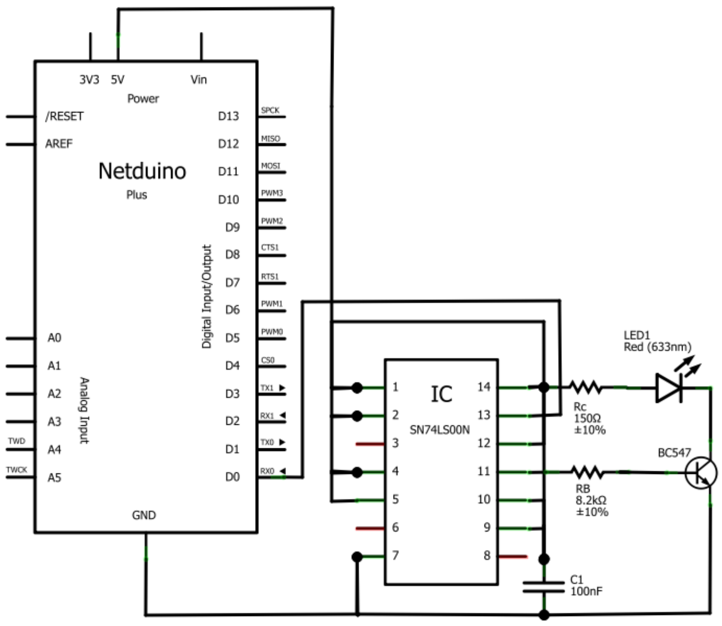
You may even end up with a variant like our third one (using second MCU pin), but also using the +3.3V to power the LED. Sounds complicated or too much? Don't worry, this just shows us that we aren't stuck with one "absolute" solution. We're free to choose between many options and whatever one we go with, it'll do the job.

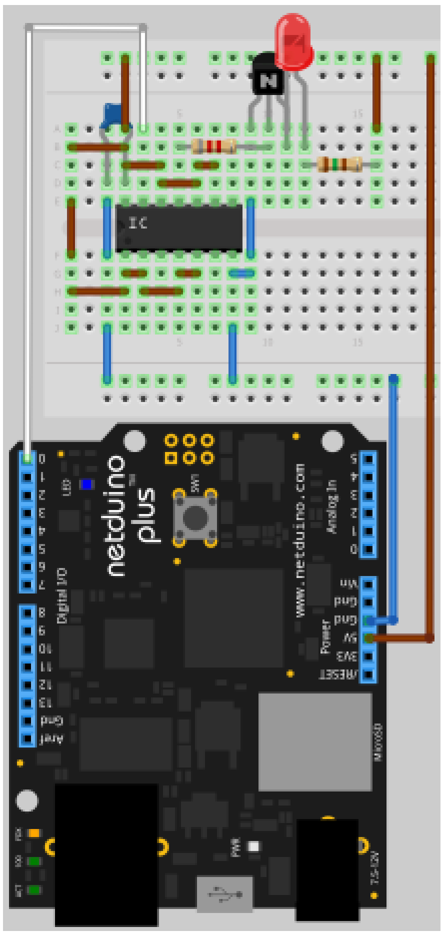
What we need now are some calculations and here we go. Remember we are going to reuse schematics from Part I, thus we have to calculate only the value of R_B. To find this value we are going to reuse the formula $(V_{OH} – V_{BEsat}) / I_{OH} = (3.4V – 0.75V) / 400µA = 6625Ω$. This time we pick one 8.2kΩ resistor because the output pin can source the maximum of 400µA. With a 6.8kΩ resistor, the current will be pretty much at the edge of this limit (mind the tolerance). Therefore, we use 8.2kΩ, which is pretty much safe (the current will be around 320 µA), and the transistor will still be fully open and in saturation mode ($V_{BE} > 0.7V$). We could also pick 7.5 kΩ, if we had one, but as it happens, I only had E12 resistors.

Where did those 3.4V and 400µA values come from, though? If you can't tell, then you'll have to go back and re-read the SN74LS00N specifications. 3.4V is the value stated as a typical one for IC output (V_{OH}) when V_{CC} is 5V and 400µA is the maximum output current (I_{OH}) that the pin can supply at high state. Note I_{OH} value is preceded by a minus sign, which indicates that current flows out of the pin. I agree with you that these convoluted datasheets are difficult to read. Unfortunately, there is no easy recipe to follow, so prepare to struggle a bit in the beginning. One excellent source of knowledge about understanding and interpreting logic datasheets can be found here.

Do we need to check the dissipated power? Sure, we can do this, but we can also skip this as some of the knowledge gathered previously tells us that it will be very little, so we don't have to worry about it at all.

Finally, all we need is in place and we can go for assembling the schematics on the breadboard. When doing this, don't forget to connect all unused IC inputs to +5V supply!





Now let's look at the application code (the full source code can be found in the "LED flash variant VII project"). In the first place, I want to mention that we can reuse code from any of the previous projects with a tiny amendment: wherever we create an instance of the LED as OutputPort we have to change the *initialState* parameter to true instead of false. Similarly, whenever we directly set the LED's status we have to change true to false and vice versa:

```
private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, false);

…

led.Write(false);
```

becomes

```
private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, true);

…

led.Write(true);
```

Although this works fine, it seems a little strange to set the program logic to false to turn ON the LED and to true to turn it OFF. It's not a big deal but a little trick will straighten the program logic back out. Inside the project (I used LED flash variant VII project), add an additional class (right click on the project name and select "Add > Class…") named PinLogic and change the code to the following one:

```
namespace LED_flash_variant_VII
{
	public static class PinLogic
	{
		public static bool True
		{
			get { return false; }
		}

		public static bool False
		{
			get { return true; }
		}
	}
}
```

Now, in the main program we can use PinLogic.True or PinLogic.False in order to set the LED status:

```
public class Program
{
	private static TimeSpan ledFlashInterval = new TimeSpan(0, 0, 0, 1);
	private static TimeSpan initialDelay = new TimeSpan(0, 0, 0, 3);

	private static Thread mainThread;

	private static OutputPort led = new OutputPort(Pins.GPIO_PIN_D0, 
PinLogic.False);
	private static InterruptPort button = new InterruptPort(Pins.ONBOARD_SW1, 
true, Port.ResistorMode.Disabled, Port.InterruptMode.InterruptEdgeLevelHigh);

	private static AutoResetEvent syncEvent = new AutoResetEvent(false);

	public static void Main()
	{
		mainThread = Thread.CurrentThread;

		// Attach button listener
		button.OnInterrupt += new NativeEventHandler(
		(uint port, uint state, DateTime time) =>
			{
```

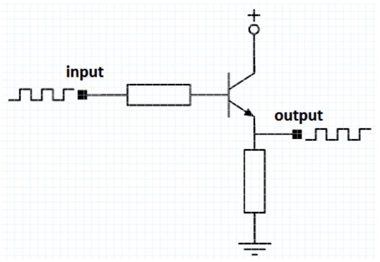
```
                            syncEvent.Set();
                    });

            // Set and start LED flashing timer
            var ledFlashTimer = new Timer(
                        new TimerCallback(
                              (object status) =>
                              {
                                    led.Write(!led.Read());
                              }),
                        null, initialDelay, ledFlashInterval);

            // Main thread waits for signal
            syncEvent.WaitOne();

            // Stop the LED flashing timer
            ledFlashTimer.Dispose();

          led.Write(PinLogic.False);
      }
}
```

Simple, isn’t it? ☺

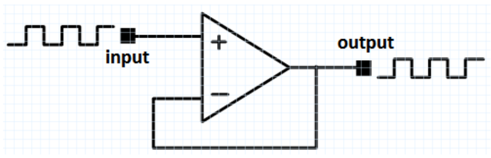
With the code in place, all concerns about this project end. However, we’ll take a closer look at some very interesting matters.

The solution we just put in place uses the current sourcing method. In this method, the current flows out of the NAND’s output. SN74LS00N can source about 400µA, which is just enough for the transistor switch to fully open. If we need more current, for example to feed more transistor switches, we need to add an additional current source.

One way to do this is to add a non-inverting transistor buffer amplifier (common collector amplifier) like the one shown on the right. By now, I’m sure that you have enough knowledge to understand how it works and how to calculate the resistor values. For a deeper understanding, you may also want to read about emitter follower (also known as common collector amplifier).

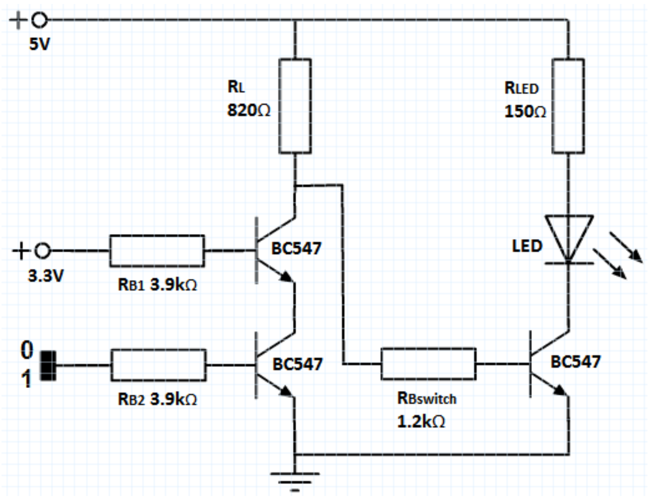


Alternatively, you can go for an IC buffer solution and turn an operational amplifier into a voltage follower like the one shown on the left.



Before we continue using IC logic gates, it’s time to have a look at how we can make a NAND gate equivalent out of BJT. Although the result will be less fancy, it will give us much more sourcing current.

The final schematic of BJT NAND with calculated element values is on the right. Try to analyse how the schematic works and consider calculating the element values for exercise.



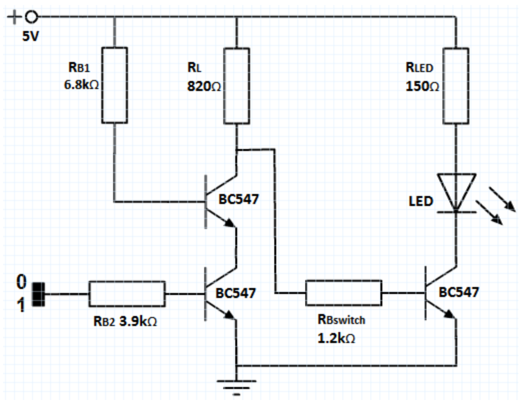
How does this work? Well, as you can see, there are two transistor switches connected in such a way so that when the MCU pin is *high* then both transistors are open and current from the +5V source flows through the R_L resistor down to the ground (sinking), thus no current can flow to the $R_{Bswitch}$ and the LED is OFF. When the MCU pin is *low*, then the current can no longer go to ground this way so it finds an alternative route through the $R_{Bswitch}$. You already know how the rest of the schematic works, so we'll only mention that the LED is now ON.

Now we'll turn our attention to the calculations. For the LED-switching part we'll make use of the schematic from Project 1 without amending it or recalculating its values. What we have to do is to satisfy the fact it requires 3.3V and 2mA on its input to light the LED. Note that this time we'll source 2mA, not 400µA!

Considering these requirements, R_L should then be R_L = (5V – 3.3V) / 2mA = 850Ω. We'll pick 820Ω, and it should be obvious to you why the voltage drop across R_L should be (5V – 3.3V) = 1.7V. Now that we have the resistor's value, we can calculate that when the two NAND forming transistors are open, the collector current (the one that will sink) will be I_C =5V / 820Ω = 6mA. Sinking 6mA is far from ideal, but we won't worry too much about this at the moment. Our goal in this part of the material is only to demonstrate a working BJT NAND equivalent of a logical IC NAND gate.

Now that we have the I_C value, we can calculate I_B for those two transistors: I_B = I_C / 10 = 0.6mA. This gives us R_{B1} = R_{B2} = (3.3V – 0.75V) / 0.6mA = 4250Ω. We'll pick 3.9kΩ to make sure that the transistors are fully open and I_C can reach 6mA.

That's all we have to do on the theoretical side, so now we can assemble the schematic and test it. You can use Netduino's 3.3V output pin. For those of you who might not like having another connection in place, you can use the schematic shown on the right. I won't give you the calculations this time because by this

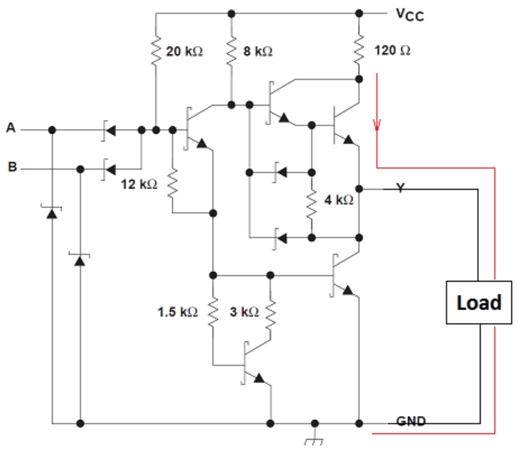


point you should be able to analyse the schematic and reach these values yourself.

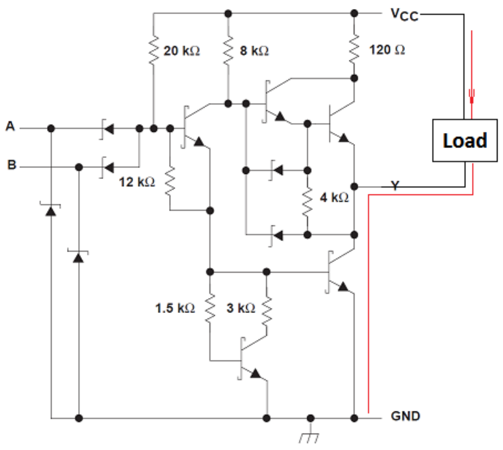
Maybe you've noticed that we're heavily reusing the building blocks we've learnt so far. Now that we understand what the principles behind those building blocks are and how they work, schematic design becomes more like LEGO. Brace yourself, as we'll continue to add more blocks.

Now let's get back to the use of ICs and current sourcing. Aside from the possibilities shown previously, let's investigate a better usage of the logic gate IC. LS chips of 7400 series can source only 400µA but can sink around 8mA, which is quite enough to light a LED in itself (unless this is a low current LED it won't be very bright, but just enough to see that it's lit). Some of the chips in this series can sink 16mA and some can sink even higher currents. So it would seem that the ability to sink current can be quite useful.

To make the principle easier to understand, follow the path (in red) of the current flowing through the load in the following examples of current sourcing and sinking with SN74LS00. I've included the full IC schematic so you can try to understand how the NAND works.



Current sourcing – result Y is high (true)



Current sinking – result Y is low (false)

By this point, you've probably noticed that the logic, at the electronic level, will be broken again (the LED will light ON MCU boot). However, there is a way to fix it: we have to use an AND gate.

Using AND gate and "sinking"

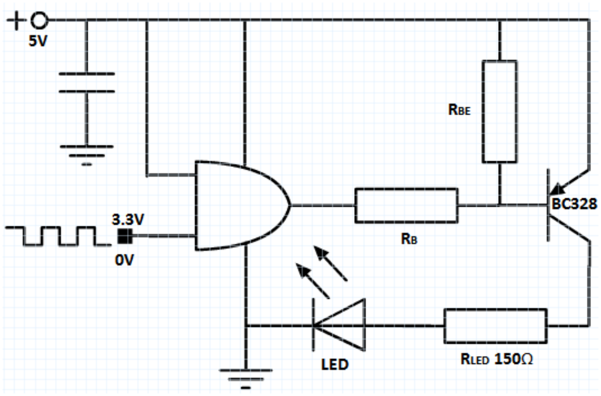
The logic will then be:

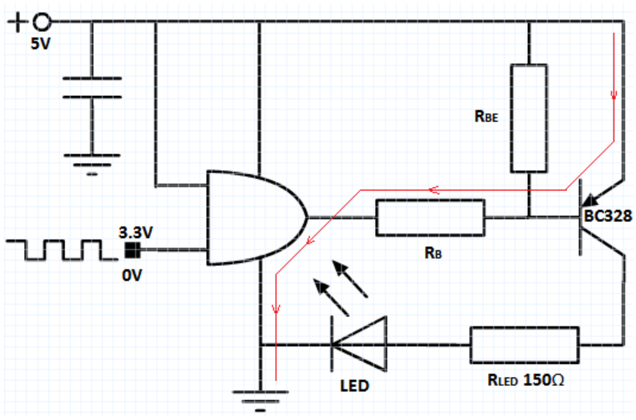
NAND input pin 1 (level is set by PSU's 3.3V/5V)	**NAND input pin 2** (level is set by MCU pin)	**NAND output pin**
High (***true***)	High (***true***)	High (***true***)
High (***true***)	Low (***false***)	Low (***false***)

With the logic correct and ready for sinking current, we need to change the transistor switch in such a way that it will switch ON when the AND gate is low. Do you remember our good friend the PNP Transistor? Remember how it works? That's right – it's open when the current flows from the emitter to the base (which is opposite to the NPN, where the current flows from the base to the emitter) and this will allow us to sink current in the AND gate's output.

Want some exercise? Stop reading here and figure out for yourself what the schematics are with SN74LS08 (TTL AND gate) and BC328 transistor switch used in "Relay approach" but instead of the relay, use a resistor and an LED. Although we can use a simpler schematic than this one, for now let's give the LEGO approach a chance and reuse the same schematics. Later we'll simplify it. Before we jump into calculations, evaluate the schematic carefully! This time I_C will be 20mA, not 24mA as it was in the case of the relay. Now do it! ☺

Your result should look the same as the one on the right. BC328, R_B and R_{BE} form the transistor switch, we replaced the relay with a LED/resistor pair and instead of connecting to the MCU output pin we connect it to the output pin of an AND gate. I also included the decoupling capacitor and the IC power connections to remind you of the need for their existence.

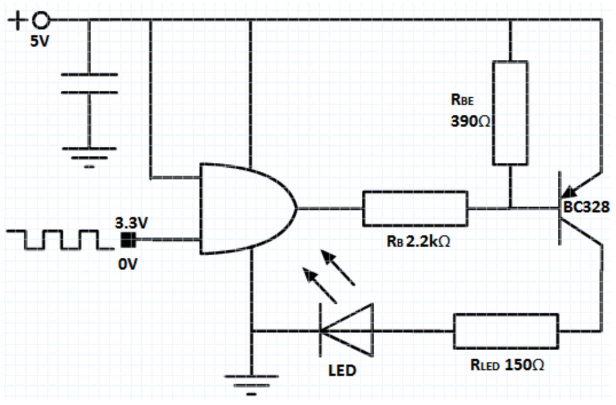




In the figure shown on the left, the red line with arrows illustrates how current sinking works when the switch is ON. This, of course, happens when AND output is *low* (*false*, 0). If you look inside the SN74LS08's datasheet, find the internal IC schematics for LS08 and follow the current path from the output pin down to ground. I think this will show you the full picture of the on-going process.

Now, can you tell what will happen when there is a *high* (*true*, 1) on the AND's gate output?

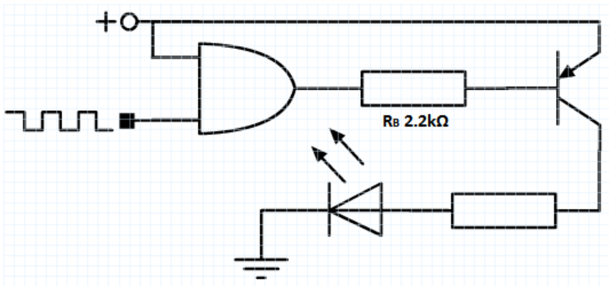
At this stage in the material, I expect you to be able to explain the processes and the result in full detail and support your analysis with some calculations. If you meet some difficulties then I strongly advise you go back to the beginning and start over. If this happens, don't let it put you off – it sometimes takes longer to understand the basics, but once this is done knowledge and wisdom will be your reward. I believe in you so it's time you believed in yourself too.



Now let's calculate R_B and R_{BE}. For R_B = (5V – 0.75V) / 2mA = 2125Ω, so 2.2kΩ seems a suitable choice. This gives us R_{BE} = ((5V x 2.2kΩ) / (5V – 0.75V)) – 2.2kΩ = 388Ω and we'll pick 390Ω.

Now assemble the final schematic shown on the right, and run the code from the previous chapter ("LED flash variant VII project").

From now on, if it's not necessary, I won't include the power connections and decoupling capacitor in the schematics. I'll take it that you learnt that lesson well and won't need to be reminded. I won't include the transistor model either, as for PNP we'll use BC547 and for PNP BC328. Instead of those two, you can use any general purpose transistor with similar characteristics. If the parameters are very different, well, you know the formulas so just recalculate the resistors.



Now, let's amend the last schematic a little by removing the R_{BE} resistor as shown on the right. Run the Netduino again. Hey, it's working!

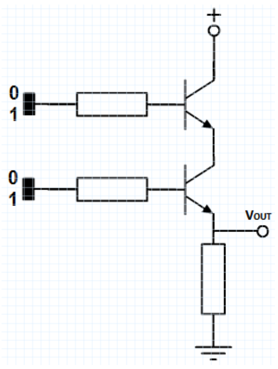
Can you explain how this is possible? It will be very beneficial to you if you can spend some time finding the answer. Use the SN74LS008 datasheet for reference of its internal schematic.

It's working because when the AND gate output pin is high we can only source current, which, where possible, will flow against the PNP transistor's emitter-base current. This means the AND's gate high output state is not opening the transistor switch. What about the potential difference between the emitter (+5V) and the AND's output (+3.4V)? If it were possible, such a current would have to flow from the PNP transistor emitter, to its base, then inside the IC. Now let me remind you of the voltage divider function in the relay schematics. In this case, the transistor's base was connected to the Netduino pin, which has an internal pull-up resistor. This resistor allows the current to flow from +5V to +3.3V potential, that is, there's enough base current to open the transistor. We put the voltage divider in place to prevent the transistor's base-emitter junction from receiving more than 0.6V, thus keeping it OFF. In the current schematic, the IC NPN transistor emitter-base junction works as a reverse connected diode, which prevents this current from flowing. In addition, 5V is the maximum base-emitter voltage that BC328 can withstand, a bit edgy but just OK. These two facts together mean that we no longer need the voltage divider and can simplify the schematic.

It's important to understand here that there are two independent factors that, in this particular case, work together in such a way as to allow us to use this improvement. Remove either one of these factors and we have to revert to using the voltage divider.

Also, note that having the divider in place did not affect the functionality of the schematic, so our LEGO approach was perfectly OK. We just improved the LEGO piece for this one concrete application.

For our final exercise of this chapter, which will be similar to the previous one, we'll have a look at the BJT equivalent of the AND gate. This example is quite interesting simply because it raises a lot of questions as well as potential problems.



Let's start with a basic schematic of the BJT equivalent, which is shown on the left. It should be easy for you to find out how this schematic works, so do that now.

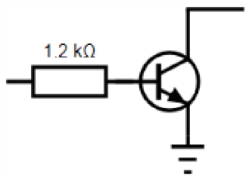
As you probably noticed, it's very similar to the BJT NAND gate, only this time V_{OUT} will be high when both transistors are open. This, of course, is due to the current flow causing a voltage drop across the resistor attached to the emitter of the lower transistor.

So what's the problem then?

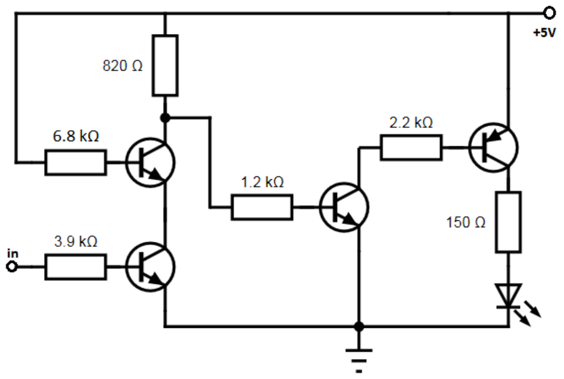
This schematic works well for current sourcing. Even in the case where the two transistors are OFF and there is a current source attached to the output pin, the resistor will cause the current to sink to the ground and nothing will stop it. This means that if we try to use this schematic for current sinking then no input will affect the output. It's useful only if we use current sourcing. However, current sourcing will make the inclusion of this BJT AND gate useless for us as it will have a high output when the inputs are high, which is exactly the case we're trying to avoid. The BJT NAND gate suffers from the same problem.

We shouldn't despair, however. We can be more persistent and insist on using BJT AND/NAND gates as a current sink, and indeed it is possible. What we have to do is to amend the schematics to have an open collector as output. Then we can connect a PNP transistor switch to it and voila, we are sinking current.

OK, so let's do it.

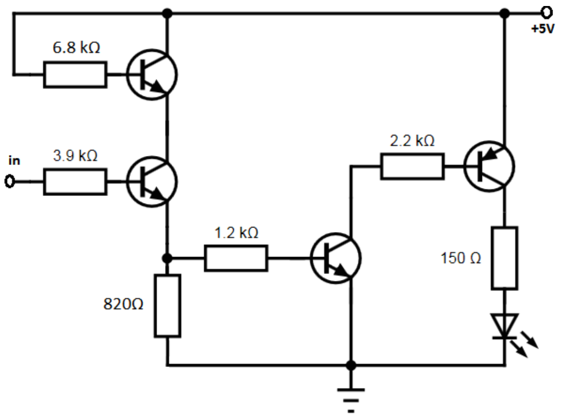


First, on the left, you can see the current sinking schematic. Looks familiar, doesn't it? Now consider how it works. If the emitter is connected to the ground and the transistor is open, then any collector-emitter current will sink.



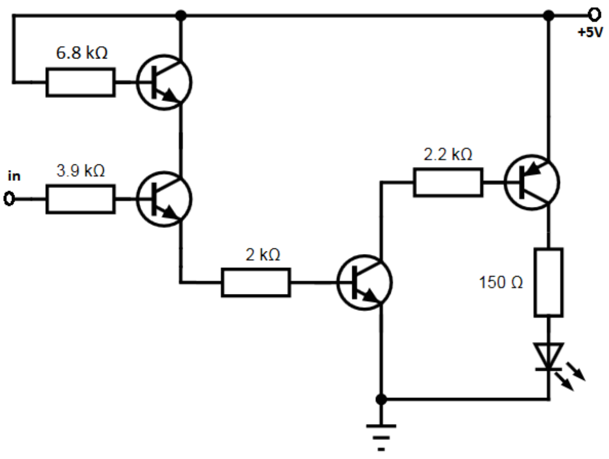
Now, following the LEGO approach, let's attach it to the NAND and also attach the PNP switch to it, with the LED from before. The resultant schematic is shown on the right.

This schematic follows NAND logic exactly, so we can now move on.



For the AND gate we have two cases. The first case is where we sink current when the logical result of the AND operation is ***true*** (1). Under normal circumstances, this is how we'll expect the AND gate to operate and the schematic looks like the one on the right.

Notice that compared to the basic BJT AND gate replacement schematic, we can eliminate the resistor here between the lower AND gate transistor and ground. The reason this transistor is there in the first place is to create voltage drop (working as current-to-voltage convertor) which we can pass on and use as the input for the next stage of the schematic. However, in the current schematic, the transistor switch can use that current by itself. Of course, it requires a small amendment to the input resistor: (5V – 0.2V – 0.75V)/2mA = 2025Ω. In case you wonder where that 0.2V drop comes from, remember that there's a collector-emitter drop across the two AND gate

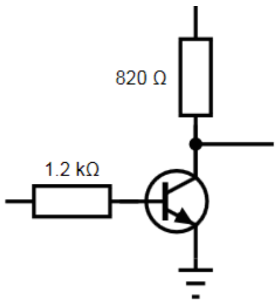


transistors, and each one is approximately 0.1V (have a look back at the BC547 datasheet). Therefore, we'll pick 2kΩ and the schematics now looks like the one on the right.

However, in our case we need the opposite result: the current sink should be OFF when the result of the AND operation is ***true*** (1) and should be ON when the result is ***false*** (0). Thus, we need to invert the result between the AND output and the current sink.

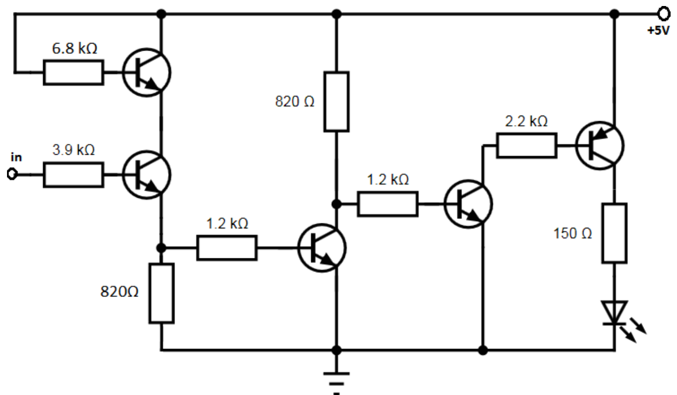
It might sound a bit complicated but actually, it isn't. Our only concern is that now we have to introduce more elements.

First, have a look on the schematic on the right.

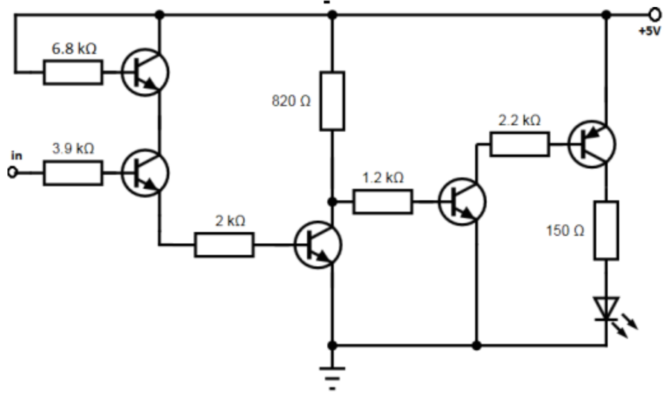


This is what the logic invertor looks like, but I'm sure you recognised it and, more importantly, that you know how it works, so nothing comes as a surprise to you here.

Now let's plug that invertor schematic between the BJT AND gate and the current sink in the previous schema. The result is shown on the right.



Again, the resistor between the lower AND gate transistor and ground isn't required, so we can remove it and thus simplify the schematic.



Wow, that's quite a lot of elements for a relatively simple logical operation. If you're unsure about using such schematics every time you need a NAND or AND gate with current sinking (add to this the last extraordinary case) I completely agree with you. Long live the ICs!

However, how do IC logic gates solve the problem with outputs being able to source and sink current? Firstly, ICs are content in having as many elements as required to implement any schematic and secondly, they use the magic of the ***totem pole***. Have a look at the internal schematics of SN7400 and SN7408 and you'll easily spot it. You'll also notice that the different ICs are using different implementations of the totem pole, but the result is the same for all of them. For now, we're not going to implement totem pole BJT replacements, but I'd suggest you design and implement at least one yourself, because it'll help you to gain a deeper understanding on the subject.

Here we're going to go back to using an IC and ask a question, which is a bit off the beaten track, but which is a good brain exercise. Remember that, in the SCR approach, we used current sinking to switch OFF the SCR? Have a look back at what we know about current sinking with the 7400 TTL series and ask yourself if it's possible to use it as a replacement for the transistor in the SCR approach with the IC one.

Unfortunately, we can't. This is because with the SCR's 5V power supply and with a maximum of 8mA to sink we'll need a 625Ω resistor to protect the IC output. This will lower the maximum current that can pass through the SCR before we lose the ability to switch OFF. Even if we use a 16mA sink, this won't be enough: 5V / 16mA = 312Ω.

Time to move on to the next logic gate.

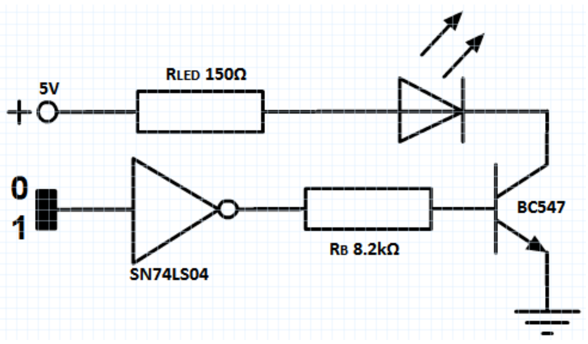
Using NOT gate

As it only has one input, a NOT gate will allow us to simplify the schematic. The output result of the NOT gate is the inverted input. We'll use SN74LS04 and you've already got all the information on how to design the schematic and write the code, so just do it.

The logic table for the NOT gate is similar to that for the NAND gate:

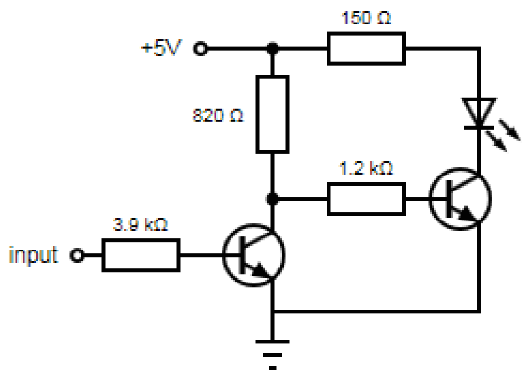
NOT gate input pin (level is set by MCU pin)	**NOT gate output pin**
High (***true***)	Low (***false***)
Low (***false***)	High (***true***)

Using the transistor switch from the NAND's approach and considering that the IC electrical parameters are the same, you should come to the schematic shown on the right.



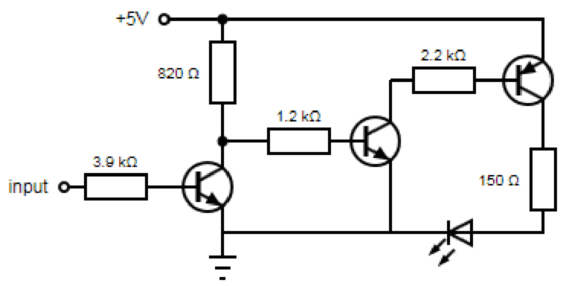
In addition to the simplified design, when using a SN74LS04 NOT gate, we can add another advantage: it has six NOT gates. Compared to the four gates that NAND and AND gates have, this give us control over more pins with a single IC.

Now let's look at the BJT equivalent of a NOT gate. In fact, we've already used it to make the BJT current sink replacement of the AND gate, remember? Well, using it with the attached switch will give you a schematic like the one on the right.



Again, this schematic is used as a current source. However, it shouldn't be a problem for you to turn this into current sink, so try to do it yourself first. If you can't see how this can be done, then go back a few pages and re-read the material.

The resultant schematic is shown on the right.



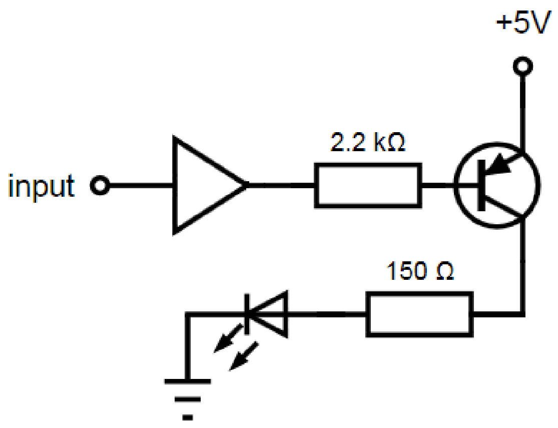
Now going back to the IC approach, we should note that we used the AND gate as the current source. The 7404 IC can be used as a current sink, but in this situation, this will break the logic again and will cause the LED to light at the boot time. This time there's no logic second pin that can help us to enforce and use a program logic trick, as we did before. However, we're not entirely helpless, as we can use a "buffer" gate, for having both required behaviour and at the same time using an IC for simplicity and as a current sink.

Using "buffer" gate

In general, a buffer gate simply displays on its output what's on its input. There are many cases when this is quite practical. Apart from our case, where we'll going to use it as a current sink, they're also very useful when you have to "feed" more consecutive inputs, as buffers are designed to sink more current than other types of gates. For example, we're going to use SN74LS07, which can sink up to 40mA. Buffer gates are usually open collector. Some buffer gates are also tri-state gates, but we're not interested in using this type of buffer, as it will cause the same problem that we're trying to solve.

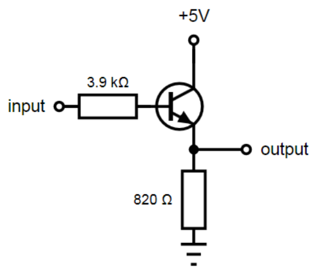
In addition, note that we can apply up to 30V to the SN74LS07's output. This means that we can control devices powered with voltages higher than the IC's power supply and in combination with a 40mA current sink, this makes the buffer gate quite a useful device.

By this point, the schematic itself, which is shown on the right, shouldn't be a mystery to you.

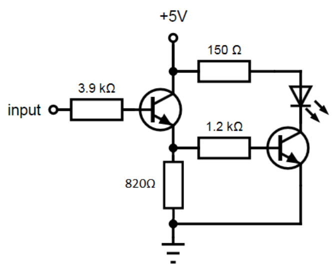


Now let's make a BJT replacement of the buffer.

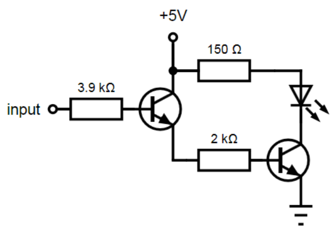
First, look at the general replacement schematic and its "evolution" to our possible solution:



buffer principle schematic

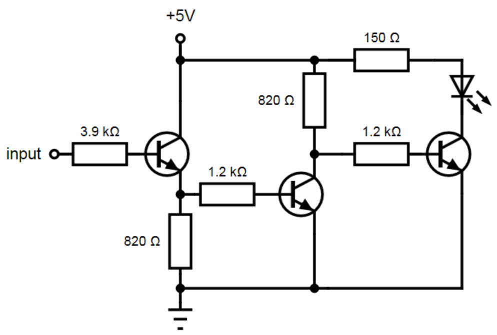


buffer with voltage drop resistor and straight logic

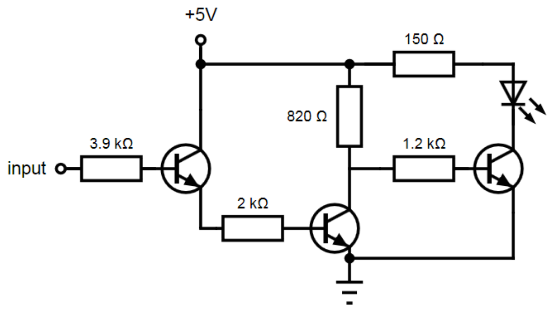


Simplified buffer with straight logic

This schematic replicates the input level to its output and this is exactly what we'll expect from buffer. However, in our case this will result with LED ON. Do you have any ideas how we can fix this? I am sure you remembered the logic invertor we used few paragraphs before, thus coming to shown on the right, current source solution.

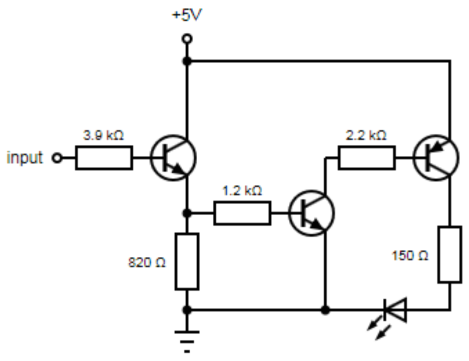


Current source buffer with voltage drop resistor and inverted logic



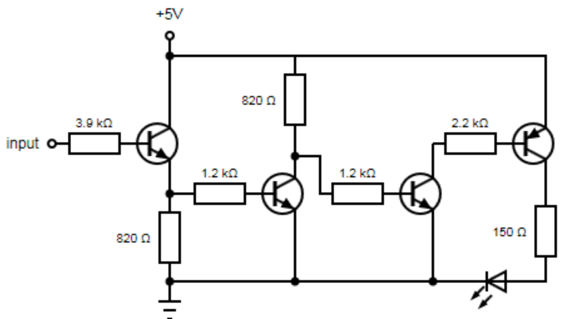
Simplified current source buffer with inverted logic

Now it's time to show the current sink buffer variations:



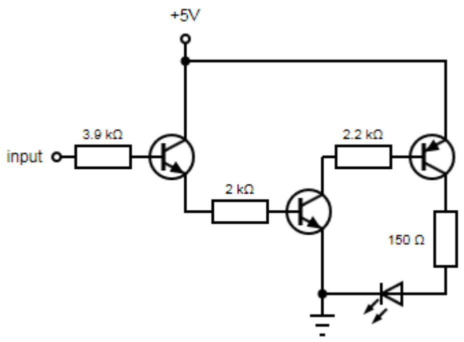
Current sink buffer with voltage drop resistor

and straight logic



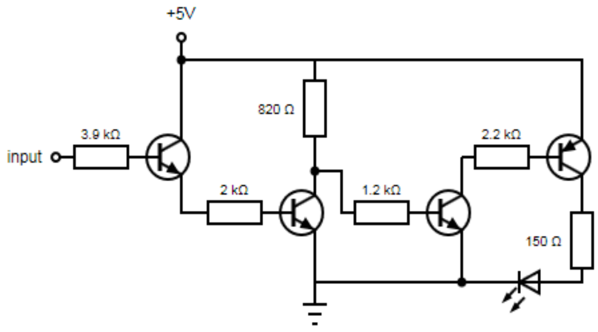
Current sink buffer with voltage drop resistor

and inverted logic



Simplified current sink buffer with straight

logic



Simplified current sink buffer with inverted

logic

Wow, there are so many schematics that your head might be spinning now. I could have avoided some of the schematics that aren't solutions to the problem we're interested in. However, I've included them just to make it easier to follow the ideas and logic behind them, as well as to show how one idea can evolve into another.

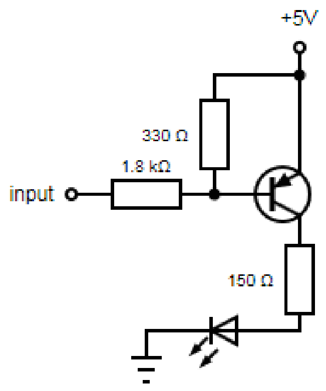
One thing you should always have in mind is that all BJT replacement schemas have higher overall current consumption compared to their corresponding IC schematic!

Before we move onto more interesting problems, let's have one more attempt and see if we can solve this problem more easily. If your reaction at this point is: "Oh, no, not again!" then perhaps you're right to be slightly bored, but my response would be: "You may be bored but you're now armed with quite a large arsenal of LEGO building blocks, and all that just from one 'simple' problem". So no more complaints, and let's make this last effort count.

Using PNP transistor with straightened reverse program logic

Finally, we come to the solution that simply uses a single PNP transistor switch. In fact, having seen the previous buffer gate approach, I'll expect you to ask the question: why do we need the buffer at all? The buffer IC is very useful if we need to sink a lot of current, so it has own niche applications. However, in our case we can avoid it.

At this point, you probably guessed that we're going to use the same schematic as we used in the relay approach, but instead of the relay, we'll power a LED. If you don't like the reversed programming logic then you can use the `PinLogic` class. The schematic is on the right.



The important thing to remember is that the following schematic is only possible because the Netduino pins are 5V tolerant! Avoid using higher than 5V to power the LED switch!

----- Future content outline -----

Make few pins control more stuff

Charlieplexing

Using demultiplexer

Using shift register

Control the flash strength

From flash to pulse

Can we do all this even better or “Use the MOSFET, Netduino…”

Make Netduino “sense” the World

Netduino, can you hear me?

Make things move

I can see you, Netduino

Netduino, can you see me?