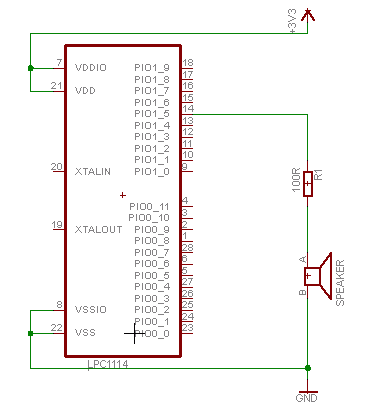
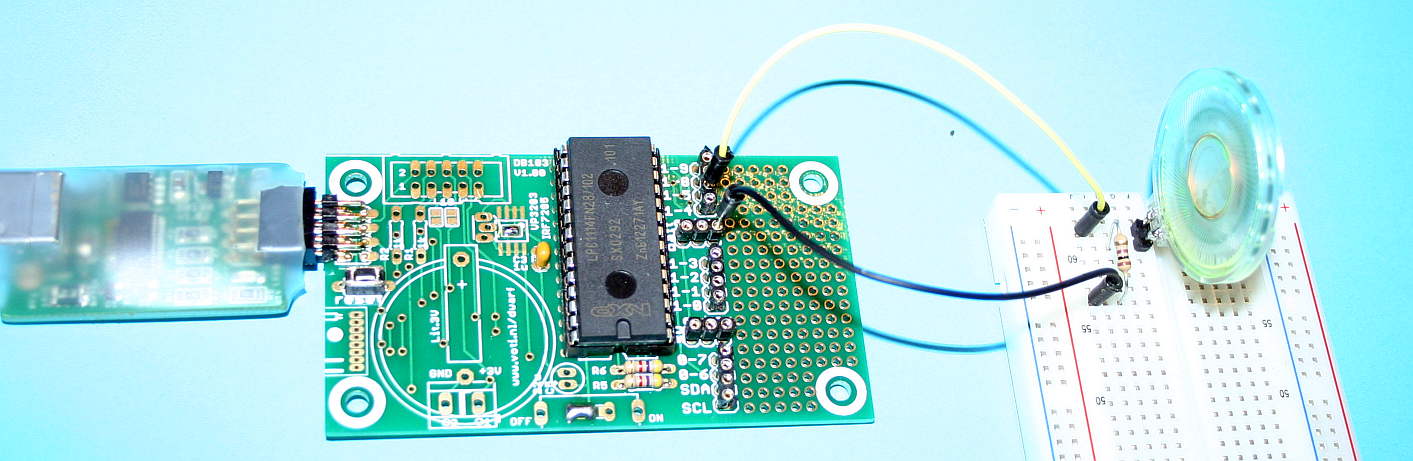
**V1TH04**

**Thema Microcontrollers**

**Reader**

****

**// make PIO1\_4 high**

**LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | 0x10**

|  |  |
| --- | --- |
|  | Wouter van Ooijen, 2014 … 2015  Hogeschool Utrecht, Technische Informatica  Version 1.1, last modified 2015-04-09 |

# Summary

The V1TH04 course teaches the use of an LPC1114 microcontroller, with the emphasis on hardware-software interfacing. It is programmed in C and simple peripheral circuits are built around it. The course consists of six lessons with fixed assignments, and a final assignment that is chosen by the student.

Table of Contents

[Summary 1](#_Toc416349825)

[1. Course organization 4](#_Toc416349826)

[2. Week 1 – blink and beep 5](#_Toc416349827)

[2.1. What is a microcontroller 5](#_Toc416349828)

[2.2. The LPC1114FN28 5](#_Toc416349829)

[2.3. Installing the development environment 7](#_Toc416349830)

[2.4. Blinking a LED - hardware 8](#_Toc416349831)

[2.5. Blink a LED - code 9](#_Toc416349832)

[2.6. Blink a LED – compiling and running 11](#_Toc416349833)

[2.7. Reading a switch 12](#_Toc416349834)

[2.8. Beep on the speaker 14](#_Toc416349835)

[2.9. Create your own project 15](#_Toc416349836)

[3. Week 2 – timing, asynchronous data format, 1-wire 17](#_Toc416349837)

[3.1. Beep at an exact frequency using a timer 17](#_Toc416349838)

[3.2. Creating and using a timer library 20](#_Toc416349839)

[3.3. The asynchronous serial data format 22](#_Toc416349840)

[3.4. An asynchronous serial library with Doxygen documentation 23](#_Toc416349841)

[3.5. Controlling a hobby servo 28](#_Toc416349842)

[3.6. IR remote control basics 30](#_Toc416349843)

[3.7. The Dallas/Maxim one-wire bus 31](#_Toc416349844)

[4. Week 3 – analog to digital (A/D) and back (PWM) 35](#_Toc416349845)

[4.1. Changing the configuration of a pin 35](#_Toc416349846)

[4.2. Analog to digital conversion 37](#_Toc416349847)

[4.3. A color sensor 41](#_Toc416349848)

[4.4. Pulse Width Modulation 42](#_Toc416349849)

[5. Week 4 – clocked communication: SPI, I2C 46](#_Toc416349850)

[5.1. Serial Peripheral Interface (SPI) 46](#_Toc416349851)

[5.2. Inter-IC communication (I2C) 48](#_Toc416349852)

[5.3. The MCP23017 I2C I/O extender 51](#_Toc416349853)

[6. Week 5 – multiplexing 54](#_Toc416349854)

[6.1. Driving a multiplexed seven-segment display 54](#_Toc416349855)

[6.2. Reading a matrix keypad 56](#_Toc416349856)

[7. Week 6 – graphics, and a small brother 60](#_Toc416349857)

[7.1. Using a Nokia 5510 graphics LCD 60](#_Toc416349858)

[7.2. Using an LPC810 chip 66](#_Toc416349859)

[Appendix A : hardware set content 68](#_Toc416349860)

[Appendix B : DB103 full circuit diagram 69](#_Toc416349861)

[Appendix C : DB103 pin assignment 70](#_Toc416349862)

[Appendix D: Nokia 5510 LCD modules connections 71](#_Toc416349863)

[Appendix E : some (wild?) suggestions for your final project 72](#_Toc416349864)

[Appendix F : the resistor color code 73](#_Toc416349865)

# Course organization

The V1TH04 course consists of 6 lectures, each corresponding with a chapter in the reader. Each week has a set of fixed assignments that are explained on the last powerpoint sheet(s) of that week. These assignments must be completed and demonstrated to the teacher, who will probably want to inspect the code written by the student. The assignments are individual. Mutual help is allowed, but the code must be your own, and you must understand your code as if it were fully yours. You can be asked to demonstrate this.

The course uses a hardware set that must be bought from the teacher for E 60. When you can find a student that has completed the course last year and is willing to sell you his set that is OK. (Note that the set is re-used in some second year courses.)

The software used in the course is free and available in a .zip file on sharepoint. Alternatively you can find the individual components on the internet. The software runs on Windows. A USB port is required to connect to the target hardware. A Linux or Mac might work, but will require some extra work to find the appropriate versions of the software packages and get it all running, which might be better spent on the real work.

For a successfully completed final assignment that (far) exceeds the expectations the student can apply for a star (mark on his list for exceptional work). Consult the teacher if you think this might apply to you.

The course has two components that you must complete: the fixed assignments and the final assignment. The fixed assignments are mandatory (without a V mark for them you won’t get your points) but don’t contribute to the final score, which is determined by the final assignment. The final assignment is chosen by the student but must be accepted by the teacher. The aim is that the assignment is sufficiently challenging for the student but still doable. It must use the LPC1114 and/or LPC810 and hardware controlled by this chip, and the student is encouraged to interface it to other hardware than the parts available in the hardware set.

This reader is meant to be used in combination with the lectures. It tries to be complete, but it might not cover everything explained in the lectures in full detail.

# Week 1 – blink and beep

## What is a microcontroller

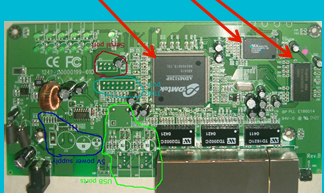
A microcontroller is a combination of all essential elements of a computer (processor, memory for code and data, Input/Output) on a single chip. The speed and amount of these elements can be very small compared to a modern PC, but they are all present. Compared to a desktop or laptop PC, or even a smartphone, the capabilities of a microcontroller are very small.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Clock frequency | RAM | ‘ROM’ | Price | Power |
| PC (anno 2014) | 3 GHz | 8 Gb | 1 Tb Disk | $ 300 | 100 W |
| LPC1114FN28 | 50 MHz | 4 Kb | 32 Kb FLASH | $ 3 | 30 mW |

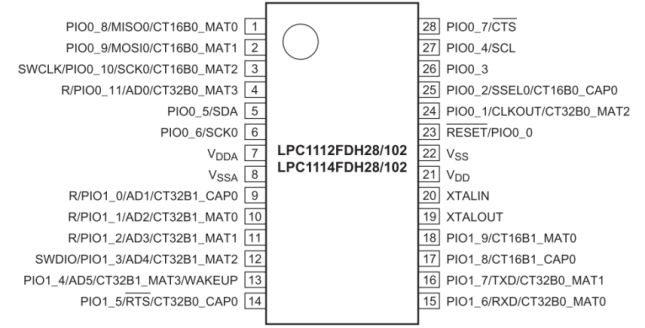
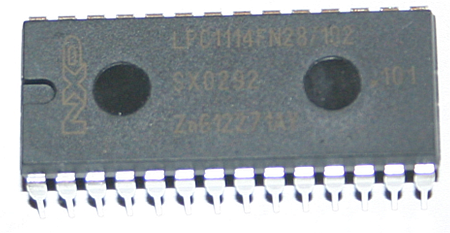
The low price, low power consumption, and small physical size make microcontrollers useable in all kinds of of products: in postcards that play Merry Christmas, thermostats, keyfobs, tamagotchi’s, wristwatches, access control systems, etc. The low power consumption is an important issue when the chip must run for long periods on battery power.



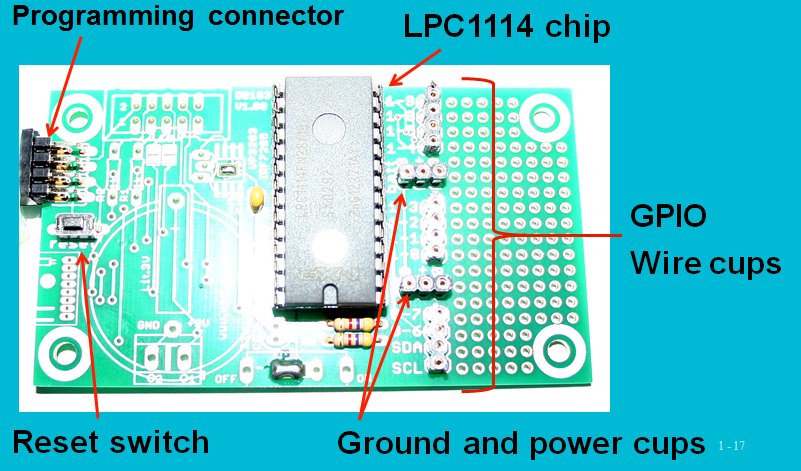
A microcontroller does not have enough ROM and RAM to run a modern OS (Operating System) like Windows, Linux, or iOS. For this separate CPU, ROM and RAM chips are required. Such a system is physically larger, and requires significantly more power, than a microcontroller. Systems like this can be found in internet modems, routers, game consoles, intelligent TVs, smartphones, tablets, and of course in desktop and laptop PCs.



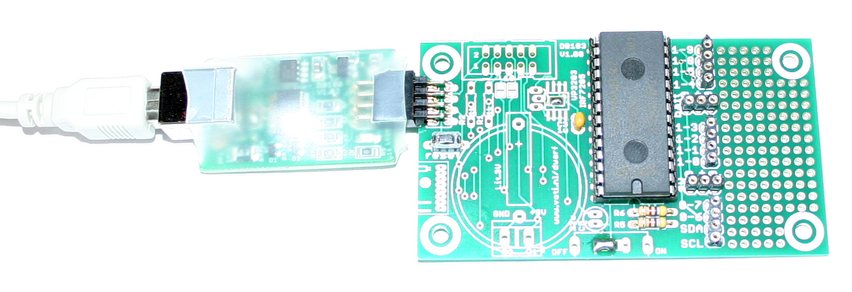
## The LPC1114FN28



The main chip we are using, the LPC1114FN28, has a Cortex M0 CPU (Central Processing Unit), which is the smaller version of Cortex A CPU’s that currently dominate the mobile gadgets markets. For our convenience we use a version in a large housing with real pins. Commercial mass-produced products might use the same chip, but in a smaller housing to save PCB (Printed Circuit Board) area (a major cost factor in mass-produced goods) and make the overall product smaller. Our chip has 4 Kb RAM (memory that can be both read and written, but loses its content when the power is removed), and 32 Kb FLASH (memory that can be written, but slowly, and only a limited number of times). The FLASH stores the application code and read-only data, the RAM stores the variables, the run-time stack, etc.



We use the LPC1114 chip in a simple PCB (Printed Circuit Board) called DB103. It contains the chip itself, a reset switch, a connector for the downloader and wire ‘cups’ for making external connections. There is room on the PCB for some other things (coin battery, power slide switch, 10-pin boxed connector, MOSFET for switching power to peripherals, RM73 radio module, and a ‘sea of holes’ for adding other components) but these are not present on the board as we use it.



The DB100 board (the smaller board, enclosed in shrink wrap) is used to

* download an application (an ‘image’) into the LPC1114 chip;
* provide 3.3 V to the LPC1114 and things we connect to it’;
* communicate with the LPC1114 for debugging and logging.

The DB100 board connects to a USB port of your computer.

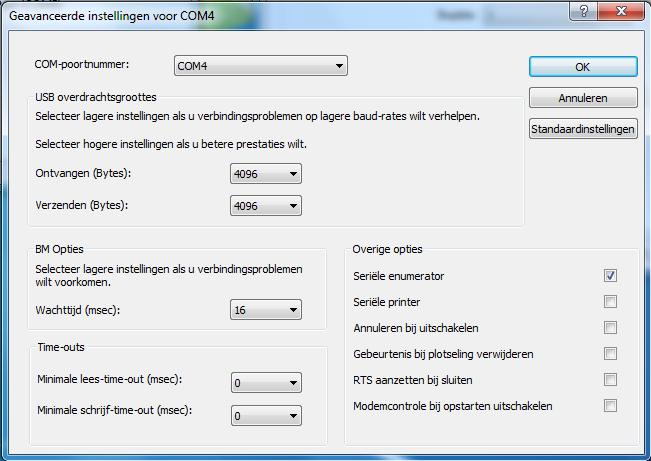
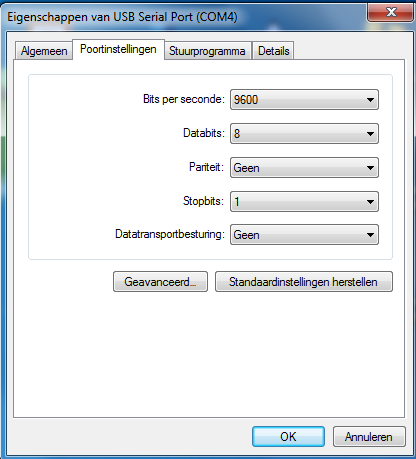
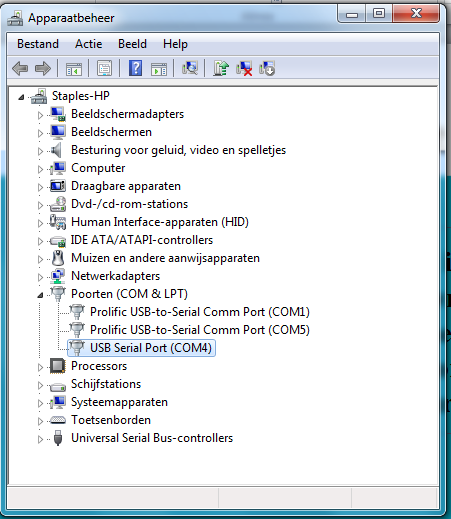
## Installing the development environment

We write code for the microcontroller in C, using an editor and compiler running on a PC. This is called cross-development: the compiler runs on one system (our PC) but produces an executable for a different system. We use the following components:

* the PsPAD editor;
* the GNU toolchain (compiler, linker and libraries) for ARM;
* the FTDI USB driver for the DB100 board;
* the bmptk makescript, and associated tools and examples.

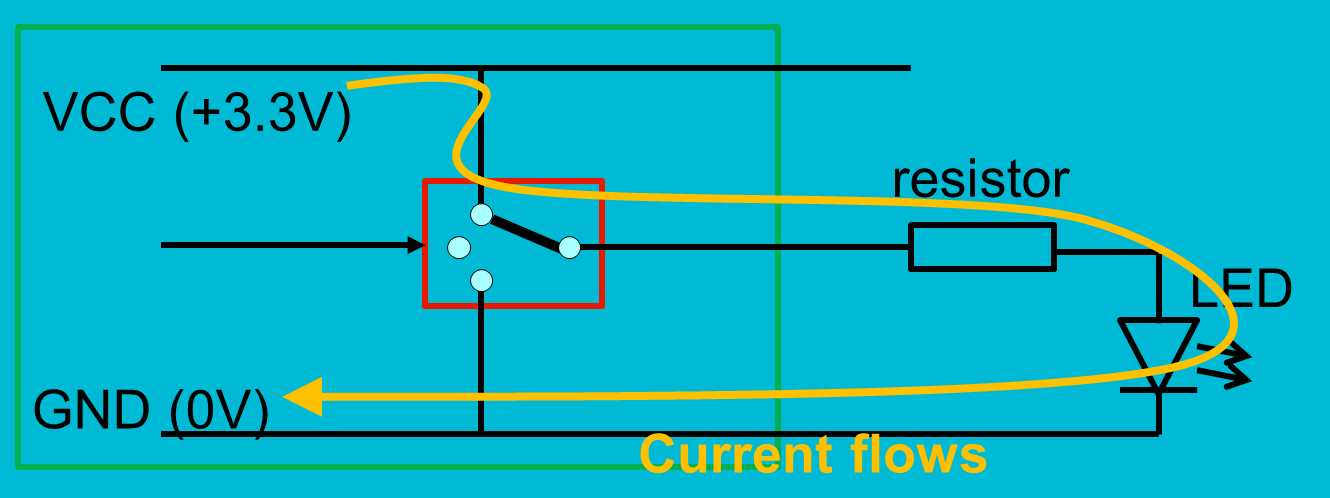
These components can all be found in the big zip file on sharepoint. Install the PsPAD, the GNU toolchain, and the FTDI drivers by running their installers. Install them in their default locations. Watch out for any bloatware the PsPAD installer will try to install on your PC. Bmptk is a simple zip file, expand it in C:\ You must add the bmptk/tools directory to your PATH variable.

Now connect the DB100 to your computer using the USB cable. After some time it should be recognized as a USB serial port. Windows will assign the next free serial port number to it, but the bmptk makefile assumes it is COM4, so you must go to the device Manager (Apparaatbeheer), select the newly created COM port (right-click on it), click Advanced (Geavanceerd), and change the COM portnumber to COM4. When you click OK you might get a message that this port number is in use, which you can ignore. Note that to your PC each DB100 is unique, so when you use another DB100 (from a fellow student) you will have to do this step again.

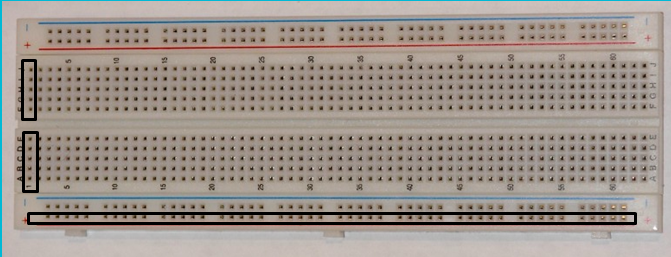


## Blinking a LED - hardware

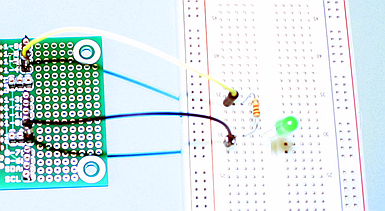
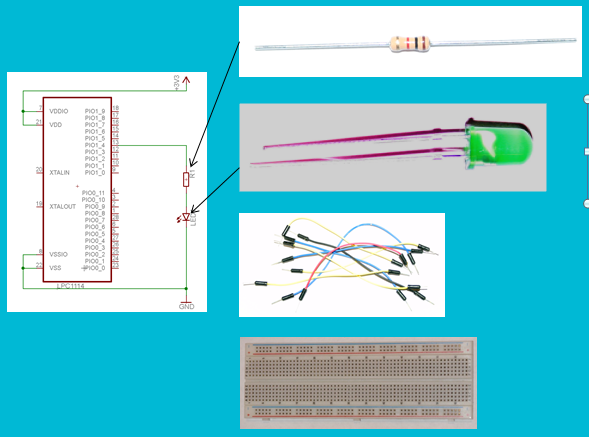
A LED is an electronic component that lights up when current passes through it in the correct direction. The current flow from the positive supply, through the switch that is part of the I/O pin circuit in the LPC1114, through a resistor, through the LED, and finally to the negative supply connection. The resistor is required to limit the current, when you omit it the LED and/or the I/O pin of the chip can be damaged.



The part in the green box is inside the LPC1114 chip. The rest (resistor, LED, and the connections) you must build. This is done on a solderless breadboard. It has holes that you can put the leads of your components and wires into. The two top rows and the two bottom rows are connected horizontally. By convention, the ones that are marked with a blue line are used for the ground supply rail, and the ones that are marked with the red line are used for the power (3.3 V) supply line. The ‘middle’ holes are connected in vertical columns of five.



For the circuit we need the DB103 board, and two components: a LED, and a 1kΩ resistor. An appendix shows the content of the hardware set, and the color codes for resistors. For 1kΩ the code is brown-black-red. Furthermore we need the breadboard and the wires to make the connections.



The circuit diagram (left part of the previous picture) shows the circuit we must build. The DB100 board has three groups of four wire cups that connect to I/O pins of the LPC1114. From these cups we use the one that is marked 1-4. The DB100 also has two groups of three cups that are marked 0+0. The middle cup of these connect to the 3.3V power (which we don’t need in this circuit), the outer ones connect to ground. The picture shows a yellow wire from this cup to a breadboard row, from there the resistor (1kΩ, color code brown-black-red) to a next breadboard row, then the LED, and finally the black wire from the LED to a ground cup. The orientation of the LED is important: the flat side (shorter wire) must be connected to ground. The resistor has no mandatory orientation.

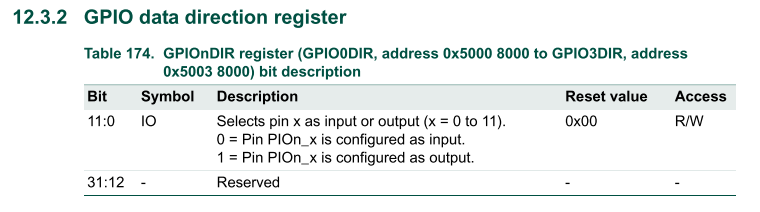
## Blink a LED - code

To make a LED blink we must switch the current on and off at the appropriate frequency. The following C program does this by first making the pin GPIO\_1\_4 an output, and then switching the pin between high and low, while waiting a little between these operations.

|  |
| --- |
| // blink a LED on pin PIO1\_4  // includes the cmsis definitions for the LPC1114  #include "bmptk.h"    int main( void ){  // must be volatile to prevent optimization of the wait loops  volatile int i;    // make PIO1\_4 an output  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | 0x10;    // loop forever  while( 1 ){    // make PIO1\_4 high  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | 0x10;  // wait  for( i = 0; i < 300000; i++ ){  }    // make PIO1\_4 low  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ 0x10;  // wait  for( i = 0; i < 300000; i++ ){  }    }  } |

The memory of our microcontroller contains the ROM and RAM, but also a large number of addresses that can be read or written by the CPU to control the peripherals of the chip. These addresses are called registers (not to be confused with the general-purpose registers in the CPU). The “#include bmptk.h” line at the start of the code includes the file (provided by the chip manufacturer) that makes these registers available as global variables. The actual chip file is LPC11xx.h, which you can find in the bmptk/targets/cortex/lpc/cmsis/11xx/inc directory.

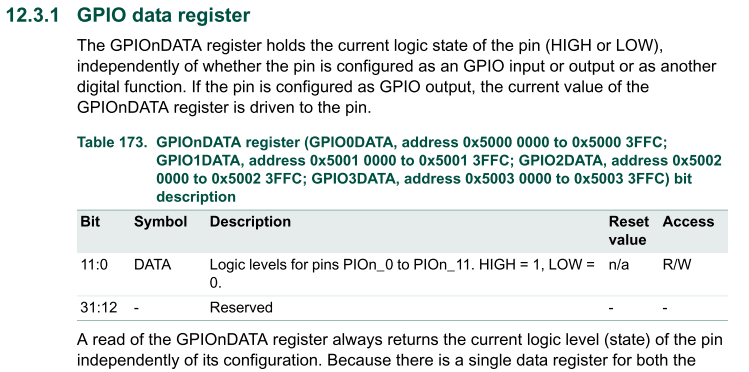
The first thing we must do is to make the I/O pin that we will use an output pin. The I/O pins can (under CPU control) be used as either input or output pins. After a reset they all default to inputs. The way to make a pin an input is described in the following fragment of the LPC1114 user manual.



It states that there are a number of registers called GPIOnDIR that control the direction of the pins, up to 11 pins per register. Such a bundle of pins is called a port. Each bit controls one pin: the pin is an input when the bit is 0, it is an output when the bit is 1. Our pin is pin 4 of port 1; hence we must make the 4th bit of GPIO1DIR a 1. It is wise not to change the direction of the other pins, hence we must read the value of the direction register, set bit 4 to 1, and write the value back. This is done in C with the bit-wise-or operator: | . The name for the register as provided by the cmsis header is LPC\_GPIO1->DIR[[1]](#footnote-1), and a value with (only) the 4th bit set is 0x10. Hence the statement for setting the 4th bit is as shown below.

|  |
| --- |
| // make PIO1\_4 an output  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | 0x10; |

Once the I/O pin has been configured as an output we can proceed to make it high. For each port there is a register that contains the bits that, for each pin that is configured as output, determines whether the pin is high (connected to the 3.3V power) or low (connected to the ground).



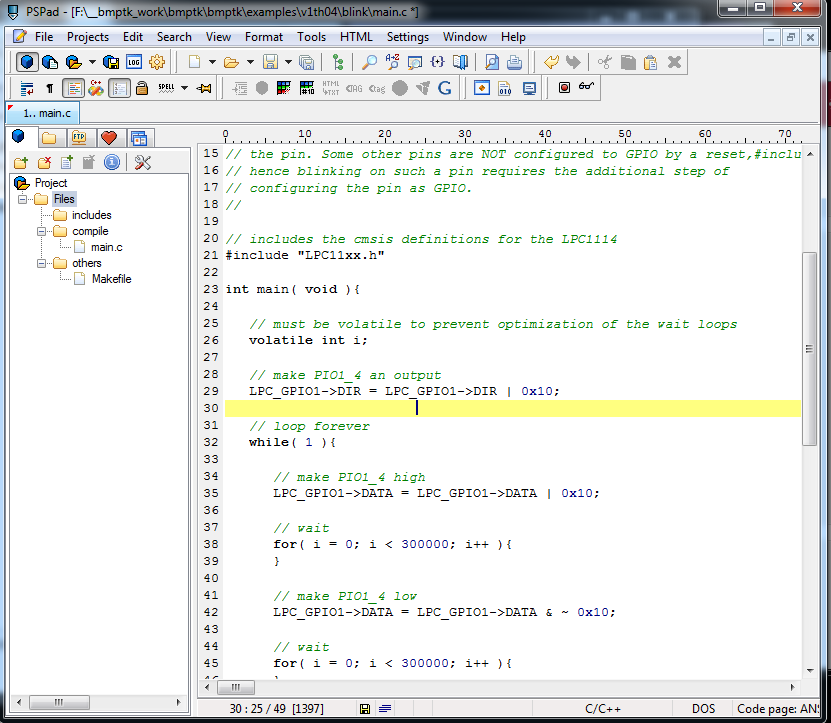
To make a pin high we must set the corresponding bit, just like we did for the direction. To make a pin low we must clear the same bit (make it a 0). This is done by logically ANDing with a value that has all bits 1, except for the bit we want to clear, which must be 0. Such a value is a bit error-prone to write down (in 32 bits is is 0xFFFFFFEF), so instead we write the opposite value (0x10) and use the ~ operator to negate the value before we use it. Hence the statement what makes the pin low is as shown below.

|  |
| --- |
| // make PIO1\_4 low  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ 0x10; |

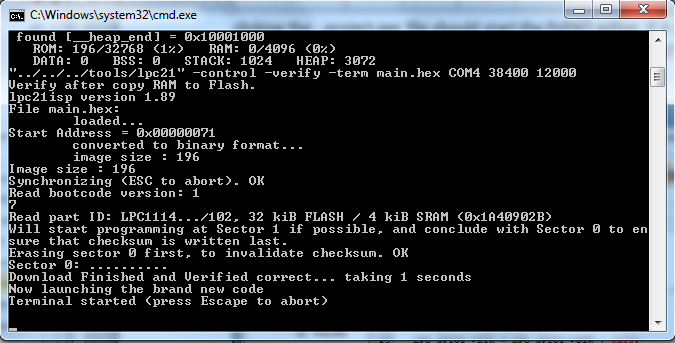
To make a LED blink we must switch the pin high and low with an appropriate delay between the switching. For this first program we use a very crude way to achieve a delay: a for loop that has an empty body but loops a large number of times. Later we will use a better method, which does not depend on guessing how many times we must loop. To prevent a clever compiler from (correctly) reasoning that an empty loop, no matter how many times it is executed, does no useful work and can hence be eliminated from the code, we must declare the loop variable as volatile. This keyword means to the compiler that each assignment to this variable is meaningful, even if it seems useless. The cmsis library declares all is variables with the same keyword, otherwise the compiler could reduce our program to nothing, because the direction and data registers are never used to do something usefull. In a sense, the only purpose of a program is to manipulate volatile variables, because that is the way to interface with the outside world. In normal (desktop) C application such code is hidden deep inside for instance the printf library code.

## Blink a LED – compiling and running

The blink-a-LED application can be found in the directory /bmptk/examples/courses/v1th04/blink. Double-clicking the \_project.ppr file should start the PsPAD editor. If it doesn’t, associate the .ppr type with the PsPAD application (which lives at C:/Program Files (x86)/PSPad editor/PSPad.exe, leave out the (x86) if you have a 32-bit Windows).

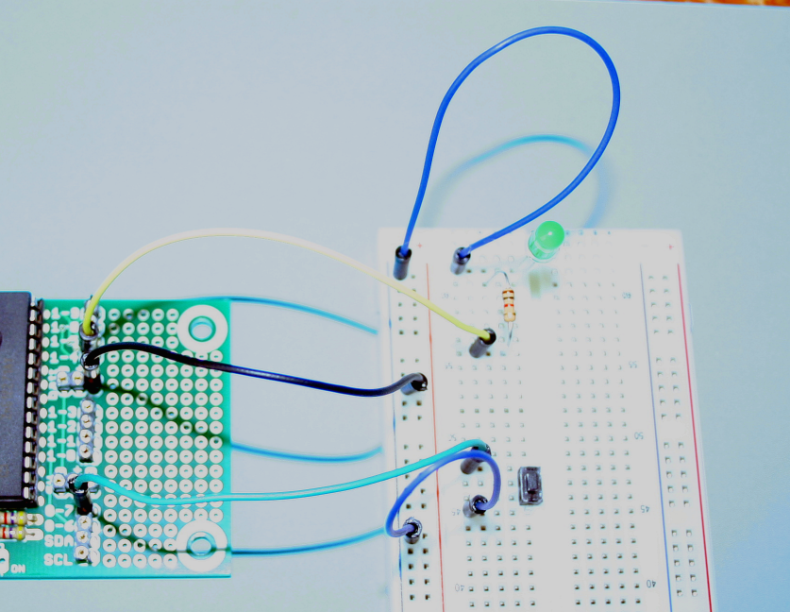


PSPad is a simple IDE (Integrated Development Environment), which means that it can start a compilation, capture the error messages, and you can click on an error message to go to the offending line in the source. The  button starts a build (compile and link) of the project in the directory of the currently open file. When building was successful, click the  button and select Run to download the compiled program to the DB100 board and start it. A command line window (DOS box) will open, which will download and start the program, and subsequently start a terminal in which you can see anything the LPC1114 chip writes to its serial port. For now you can ignore this and close the window with ESC.



## Reading a switch

A GPIO pin of the LPC1114 can be used as an output (as we did to blink a LED) or as input. The by default the GPIO pins are input, and by default a weak pull-up resistor inside the LPC1114 is enabled which makes the input pin high if we don’t pull it low externally. This can be done with a switch, which is a contact that is normally open, but closes when we push the button. We can connect such a switch between the GPIO pin and the ground, as shown in the circuit diagram and photo below.



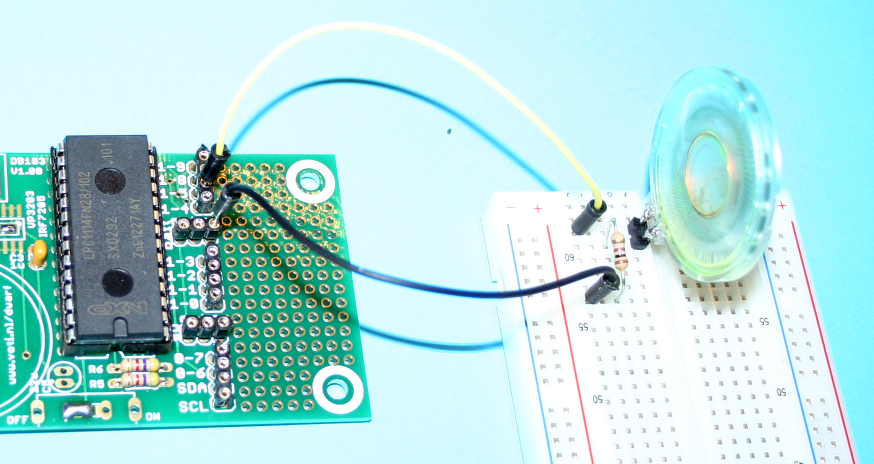
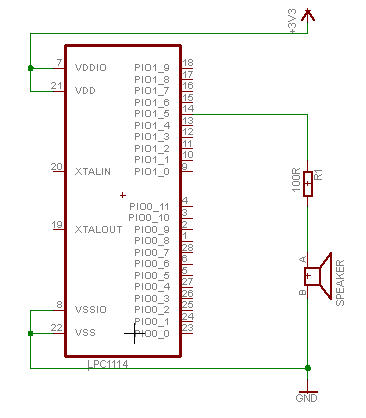
The example code below continuously copies the value that it reads from the switch (pin 0\_7) to the LED (pin 1\_4). Instead of putting the pin access code in the main() it uses three functions, one to initialize the LED pin to output, one to set the LED pin to a specified level, and one to read the switch pin. This way the details of how a pin is accessed are split from the logic of the program, and both aspects could be re-used in a different context.

Note that the magic values like 0x10 are replaced with expressions like ( 0x1 << 4 ). To the compiler this is exactly the same (as constant expression is calculated at compile time), but it makes the code better readable: The value 0x10 does not immediately reveal that it is the 4th GPIO that is used.

|  |
| --- |
| #include "bmptk.h"  void led\_init(){  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | ( 0x1 << 4 );  }  void led\_set( unsigned char x ){  if( x ){  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | ( 0x1 << 4 );  } else {  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ ( 0x1 << 4 );  }  }  unsigned char switch\_get(){  return ( LPC\_GPIO0->DATA & ( 0x1 << 7 ) ) == 0x0;  }    int main( void ){  led\_init();  for(;;){  led\_set( switch\_get() );  }  } |

## Beep on the speaker

Beeping is just blinking, but much faster, and using a speaker instead of a LED. We use a lower-valued resistor (100 Ω, color code brown-black-brown); otherwise the sound will be too weak. Unlike a LED, a speaker has no direction (both connections are equal).



The code for the beep program shown below uses a beep() function with two arguments: the number of cycles, and the delay between each toggle of the output pin (the number of iterations of the for loop). Instead of looping forever, the main() loops three times, beeping on two frequencies.

|  |
| --- |
| #include "bmptk.h"  void beep( int n, int x ){  volatile int i;  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | ( 0x1 << 5 );    while( n > 0 ){  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | ( 0x1 << 5 );  for( i = 0; i < x; i++ );    LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ ( 0x1 << 5 );    for( i = 0; i < x; i++ );    n--;  }  }    int main( void ){  int i;  for( i= 0; i < 3; i++ ){  beep( 500, 500 );  beep( 400, 800 );  }  } |

## Create your own project

The bmptk directory has the code of the examples in the bmptk/examples/courses/v1th04 directory. The easiest way to create your own project is to make a directory at the same level as the examples, for instance bmptk/work/v1th04/week1/blink for the first assignment. In that directory you can edit the file(s). For Now, take care to keep all files for a project in the same directory, and watch out fotr accidentally opening a file in the editor in another directory.

|  |  |
| --- | --- |
|  |  |

If you want to put your work in a different place you will have to edit the Makefile. The BMPTK := line of this file specifies the location of the root of the bmptk. For the examples this is specified relatively:

|  |
| --- |
| # Specify the location of the bmptk library  BMPTK := ../../../.. |

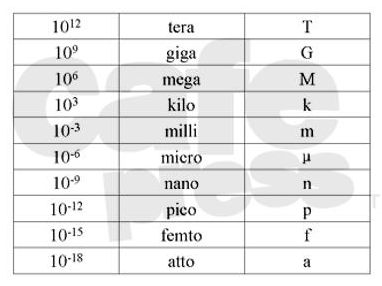
When you place your work outside the bmptk tree it is easier to specify the bmptk location absolutely, for instance:

|  |
| --- |
| # Specify the location of the bmptk library  BMPTK := C:/bmptk |

# Week 2 – timing, asynchronous data format, 1-wire

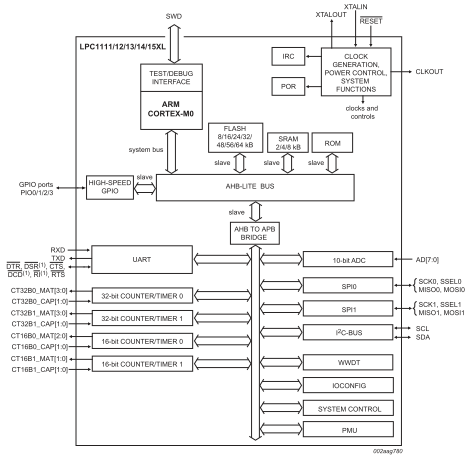
## Beep at an exact frequency using a timer

Frequency is how often something happens. The unit of frequency is Hz, which is the same as ‘per second’. In practice things can happen many times each second, so we use kHz (1000 or 10^3 times per second), MHz (1000\_000 or 10^6 times per second) or GHz (1000\_000\_000 or 10^9 times per second). Humans can hear sounds of roughly 40 Hz .. 10 kHz. Our LPC1114 chip runs at 12 MHz, executing somewhere near that number of instructions per second. The CPU in a modern PC runs at 3 GHz, but because such a CPU is one the one side multi-processor and multi-issue, but on the other hand it must often wait for one of its many memories, it is difficult to translate that to the number of instruction it will execute per second.

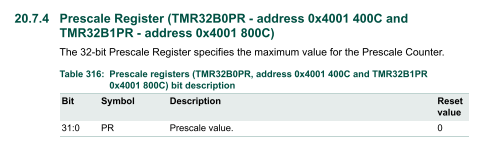
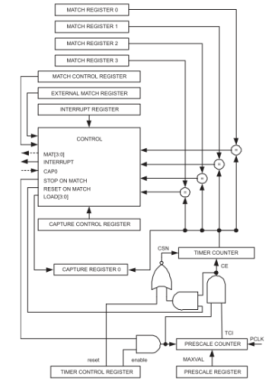


Note that for periodic signals the frequency refers to the number of periods per second. For a blinking LED for instance, a period consists of an ON phase and an OFF phase. Hence blinking at 1 Hz requires a delay (between switching on and switching off) of half its period, hence of 500 ms (m = milli = 1/1000).

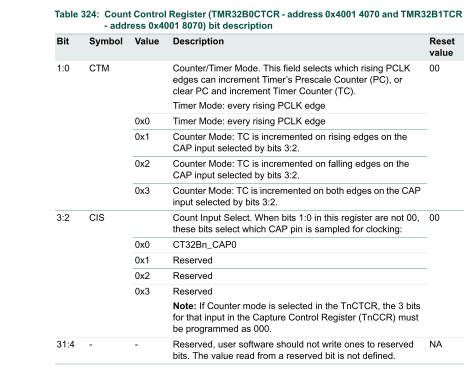
Last week’s beep code used a for loop as delay. This can work, but the uncertainty in the delay such a loop causes is large. The best we can do is measure how long it actually takes, and modify the number of iterations until the delay is as long as we want. But when install the next version of the compiler, or use a different optimization level, or maybe even when we change something in another part of our program, the delay can change. Clearly we need a better way to create a delay. The LPC1114 provides the hardware support for this in the form of a number of hardware timers. These are (16, 24 or 32 bit) counters that can be incremented automatically at a frequency that can be derived from the processor clock. The CPU can write and read the counter in a timer. If we want a delay of 50 ms, and we can arrange for a timer to count down at 1 MHz, write 50\_000 to it, and wait for it to reach 0. The LPC1114 has 2 general-purpose 32-bit timers and two general-purpose 16-bit timers (show below), and a 5’th one (24 bit) that is more limited.



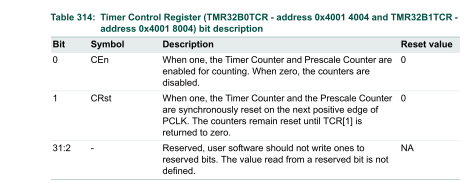
Each timer has a set of associated registers that determine what the timer does. The Counter Register does not receive the CPU clock directly, instead the CPU clock is first fed to the Prescale Counter. This is a count-down register. When it is 0 and gets a clock pulse it gives a clock pulse to the counter register proper, and it is loaded with the value of the Prescale Register. Hence the effects that the CPU clock is divided by the value in the Prescale Register (plus one) before it is presented to the counter register.



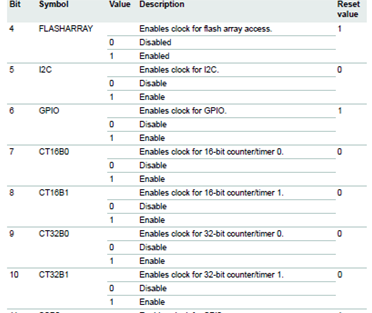
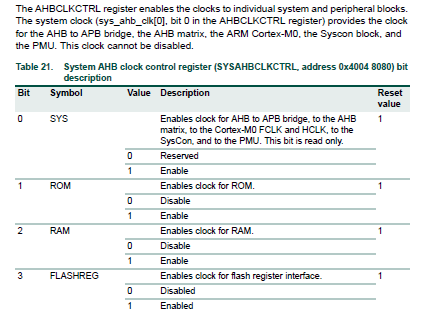
The counter can be configured to take its input frequency from a number of sources. We want to use the CPU clock (the PCLK), which happens to be the default.



By default a timer is disabled. To enable it, we must write a 1 to the CEn bit in the Control Register.



The LPC1114 chip by default disables the clock to most of its on-chip peripherals to save power. The table below shows that this is the case for all 4 general-purpose timers. To enable the clock to the time we will use (CT32B0, the first 32-bit timer) we must make the applicable bit in the SYSAHBCLKCTRL register a 1.



The code below uses the TMR32B0 to produce a beep of exactly 440 Hz. (The accuracy is determined by the accuracy of the LPC1114’s internal clock, which is 1% or better.) It first initializes the GPIO\_1\_6 pin and the counter (with prescaler 0, hence the counter counts at 12 MHz). Then it gets the value of the counter, adds the number of ticks it wants to wait, and waits until the counter has reached or surpassed that value. (Note that it is not wise to check for equality, because the while loop might not sample the counter often enough to catch it at exactly that value.)

Note that the SYSAHBCLCKCNTRL register is updated using the |= operator. This operator lets us express an operation of the form ‘a = a | b’ without writing the ‘a’ part twice. (Compare it to the line above it.) This shorthand is available for all binary operators.

|  |
| --- |
| #include "LPC11xx.h"  #define PERIOD 27273    int main( void ){  unsigned int i, next;    LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | 0x20;    LPC\_SYSCON->SYSAHBCLKCTRL |= 1 << 9;  LPC\_TMR32B0->CTCR = 0x00;  LPC\_TMR32B0->PR = 0;  LPC\_TMR32B0->TC = 0;  LPC\_TMR32B0->TCR = 0x01;  next = LPC\_TMR32B0->TC;  for( i = 0; i < 440; i++ ){    next += ( PERIOD / 2 );  while( LPC\_TMR32B0->TC < next );  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | 0x20;  next += ( PERIOD / 2 );  while( LPC\_TMR32B0->TC < next );  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ 0x20;  }  } |

The advantage of this technique of adding the delay to the current time, instead of waiting for a certain amount of time, is that the other things we must do in the loop (making the pin high and low, and counting the number of loop iterations) will (provided that they don’t take too long) have no influence on the timing. A disadvantage is that this will only work until the 32 bit timer overflows. At 12 MHz a 32 bit timer will overflow in 358 seconds, so in this simple form this technique cannot be used for serious programs.



## Creating and using a timer library

In the previous program the use of the counter was all over the main(). This is fine for a first test, but to make using a timer easy it is better to put the details in a library. A simple library consists of a .h file that contains (only) the function definitions, and a .c file that contains the implementations. The timer.h file shown below provides three functions:

* timer\_init() must be called first to initialize the timer.
* now() returns the number of microseconds (µs) since the timer\_init() call as an unsigned integer.
* await(t) waits until the time t, in microseconds since timer\_init(), has arrived (or passed).

|  |
| --- |
| // file timer.h  void timer\_init( void );  unsigned int now();  void await( unsigned int t ); |

The implementation in the timer.c file is much like what we saw in the main()-only program. In the initialization the Prescale Register is set to 11, hence the CPU clock (12 Mhz) is divided by 12 to get one tick per microcseconds. The now() function simply returns the current value of the timer’s Counter Register. The await() function loops until the now() function returns a value that it larger than the time it was asked to wait for.

|  |
| --- |
| // file timer.c  #include "timer.h"  #include "LPC11xx.h"  void timer\_init( void ){  LPC\_SYSCON->SYSAHBCLKCTRL |= 1 << 9;  LPC\_TMR32B0->CTCR = 0x00;  LPC\_TMR32B0->PR = 11;  LPC\_TMR32B0->TC = 0;  LPC\_TMR32B0->TCR = 0x01;  }  unsigned int now(){  return LPC\_TMR32B0->TC;  }  void await( unsigned int t ){  while( t > now() ){  }  } |

With the timer library the 440 Hz program can be rewritten as below. All timer details are now deferred to the timer library. Note that the PERIOD is now derived by dividing 1 MHz (the tick frequency of the timer) by 440 Hz (the frequency we want), and that the time between setting and clearing the output pin is half the period.

|  |
| --- |
| #include "LPC11xx.h"  #include "timer.h"  #define PERIOD ( 1000 \* 1000 / 440 )    int main( void ){  unsigned int i, next;  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | 0x20;  timer\_init();  next = now();  for( i = 0; i < 440; i++ ){    next += ( PERIOD / 2 );  while( now() < next );  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | 0x20;  next += ( PERIOD / 2 );  while( now() < next );  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ 0x20;  }  } |

The Makefile, which is in the project directory, specifies which source files are part of a project. By default the source file that matches the project name (which is by default ‘main’) is part of the project. If we want other files to be part too we must specify them as shown below.

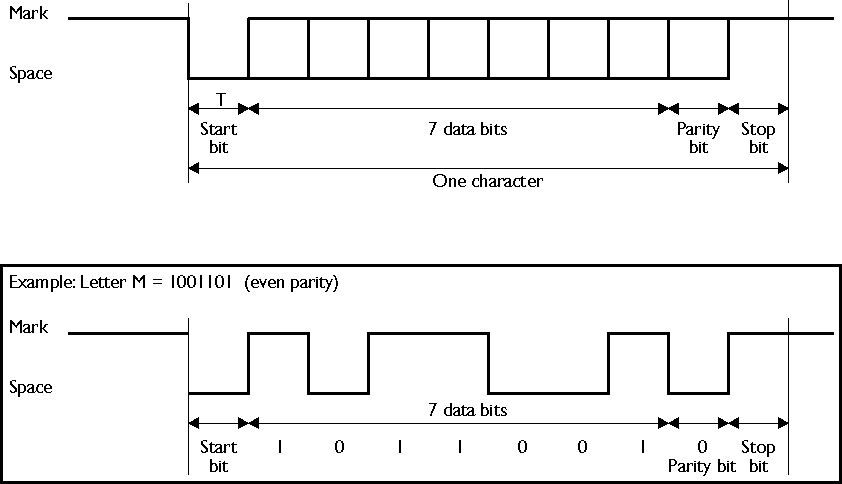
|  |
| --- |
| # Specify target chip or board or system  TARGET := lpc1114fn28  # Specify the location of the bmptk library  BMPTK := ../../../..  # Specify project files (other than $(PROJECT).cpp) (if any)  SOURCES := timer.c  HEADERS := timer.h  # The Makefile.inc does all the work  include $(BMPTK)/Makefile.inc |

## The asynchronous serial data format

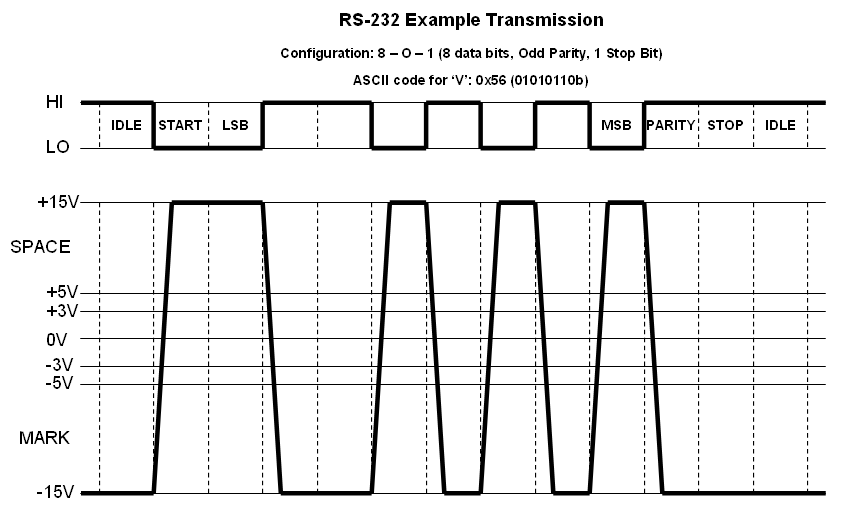
The asynchronous serial data format (also called the UART format) is a simple way to transmit a byte over a serial line. The principle is:

* the line is high (1) when there is nothing to send
* when a byte is to be sent one 0 cell is sent first (this is called the start bit)
* then all eight bits are sent, one cell per bit, lowest bit first (the example below shows 7 data bits, followed by one parity bit)
* finally one (or sometimes two) 1 cells are sent
* a cell has a fixed length; the number of cells per second is called the baudrate.

The example below shows the transmission of the capital M. According to the ASCII code (the common way to transmit characters) the bit pattern must be binary 1001101 . The lowest bit is transmitted first (that was an arbitrary choice, but once made we have to stick with it), and it is prefixed with a 0, so it is transmitted as 01011001. In the example 7 data bits are transmitted followed by one parity bit, which was once the common format. We will use 8 data bits and no parity bit, which more common now.



The 1 and 0 levels are officially called Mark and Space. Between chips Space is 0 and Mark is 1 (3v3 for our chip). For longer distances the RS232 levels are used, which are higher, and inverted: Space is 3 .. 15 V, and Mark is -3 .. -15V. (We will not use these levels.)



## An asynchronous serial library with Doxygen documentation

The examples/v1th04-sw-uart directory contain library code that can be used to send a character over the serial line to the PC to be displayed in the lpc21isp terminal window (that black window that you used to close immediately). The uart.h file contains the functions that are available to the user of the library, along with the documentation of these functions in Doxygen format. Note that functions and other declarations that are

The Doxygen lines all start with ‘//!’. The first line is required to inform Doxygen that this file contains Doxygen comments. The next lines are the general description of the library. The documentation before each function consists of three parts:

* A short single-line description of the function’s purpose. This line is optimized for information content and briefness, it is NOT a syntactically complete sentence with a Captialized first word, a noun, and ended with a period. Its intention is to let a potential user know what the function does, so he can decide whether he wants to use it.
* A // comment line without ! , to separate the short description from the long description.
* A long description, which gives sufficient information to the user to use the function. This includes a description of the parameters and prerequisites (for instance initialization that must be done first). The long description consists of full sentences.

|  |
| --- |
| //! @file  //! This library implements a bit-banged output-only UART at  //! the baudrate specified by BMPTK\_BAUDRATE.  //! initialize the UART  //  //! Call this function to initialize the UART pins.  void uart\_init( void );  //! write a single character  //  //! This function writes a single character using the standard UART  //! output pin of the chip (GP1\_7) and the baudrate BMPTK\_BAUDRATE,  //! which is passed as a command line argument during compilation.  void uart\_put\_char( char c );  //! write a character string  //  //! This function writes the characters of a standard  //! 0-terminated string  //! using uart\_put\_char().  void uart\_put\_string( char \*s );  //! write an integer in decimal format without leading zero's.  //  //! This function writes an integer in decimal format, without  //! any leading zero's, using using uart\_put\_char().  //! A negative value will have a '-' in front of it.  void uart\_put\_int\_decimal( int x );  //! write an unsigend integer as n hexadecimal digits  //  //! This function writes an unsigend integer as n  //! hexadecimal (0-F) digits.  //! Digits higher than n will be ignored.  void uart\_put\_int\_hexadecimal( unsigned int x, int n ); |

The Doxygen tool is invoked from the command line, generally in the directory where the source files are. It requires a Doxyfile, which specifies details about how the documentation is to be generated. For our purposes the file in the examples/v1th04/sw-uart directory is sufficient.

Running the doxygen command in the directory that contains the Doxyfile and header (.h) files with Doxygen comments generates a docs subdirectory that contains html documentation. When you open the index.html file you can reach the sw-uart page by selecting Files, then sw-uart.h.

The software uart implementation is shown below. It uses the timer library, hence the timer.h file is included. The BMPTK\_BUADRATE is a macro that is provided by the compilation system. It is the baudrate that is used for both downloading the application to the LPC1114 chip, and for the terminal that is started after downloading, hence it is also the baudrate we must use to communicate with the PC.

The txd\_low() and txd\_high() functions set the line low and high in the usual way. The uart\_init() makes the line an output and sets it to the idle level (high). uart\_put\_string() simply calls uart\_out\_char() for each character in the string.

|  |
| --- |
| **#include "LPC11xx.h"**  **#include "uart.h"**  **#include "timer.h"**  **#define BIT\_TIME ( (1000 \* 1000) / (BMPTK\_BAUDRATE) )**  **void txd\_low( void ){**  **LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ ( 1 << 7 );**  **}**  **void txd\_high( void ){**  **LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | ( 1 << 7 );**  **}**  **void uart\_init( void ){**  **LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | ( 1 << 7 );**  **txd\_high();**  **}** |

The real work is done in the uart\_put\_char() function. It starts by sending the start bit: the line is made low, and await() called to wait one bit cell.

Next the for() loop sends the 8 bits in the character. The if statement tests if the lowest bit is 1 or 0 by AND-ing the character with 0x01, and testing whether the result is 0. If so, the lowest bit was 0. The line is set accordingly, and await() is called to wait one bit cell. Meanwhile, to prepare for the next iteration of the for loop, the character is shifted one position to the right, so what previously was the 2nd lowest bit is now the lowest bit, which will be tested by the if statement. This way all 8 bits are transmitted.

Finally the line is made idle (high) for two bit cells and that is it.

|  |
| --- |
| **void uart\_put\_char( char c ){**  **unsigned int t = now();**  **int i;**    **// start bit**  **txd\_low();**  **t += BIT\_TIME;**  **await( t );**    **for( i = 0; i < 8; i++ ){**  **// one data bit**  **if( ( c & 0x01 ) == 0 ){**  **txd\_low();**  **} else {**  **txd\_high();**  **}**  **c = c >> 1;**  **t += BIT\_TIME;**  **await( t );**  **}**    **// stop bits**  **txd\_high();**  **t += 2 \* BIT\_TIME;**  **await( t );**  **}** |

Now that we can send a single character it is relatively easy to send other things. uart\_put\_string() simply calls uart\_out\_char() for each character in the string (ending at the ‘\0’ chacacter that signals the end of the string, not sending it).

|  |
| --- |
| **void uart\_put\_string( char \*s ){**  **while( \*s != '\0' ){**  **uart\_put\_char( \*s );**  **s++;**  **}**  **}** |

To print an integer value in decimal format is a bit complicated because

* we must handle negative values, and
* the negative rage has one more value (INT\_MIN) than the positive range, and
* the digits are calculated right (lowest) first, but must be printed left (highest) first.

A naïve solution to the negative values problem would be to check if the value is negative, and if so print a ‘-‘, convert the number to positive, and then proceed with printing the value which now must be positive. But this fails because the 2’s complement notation (used in virtually all CPUs today, including the one in our LPC1114) can represent one more negative value than it can represent positive values. Hence we can’t convert all negative values to positive. But the reverse is possible: we can convert all positive values to negative. The uart\_put\_int\_decimal() function checks whether the value negative, if so it prints a ‘-‘ character, if not it converts the value to negative. Finally it calls the uart\_put\_int\_negative() helper function, who prints a negative value, but as if it were positive. It will never be called with a positive value.

|  |
| --- |
| void uart\_put\_int\_decimal( int x ){  if( x < 0 ){  uart\_put\_char( '-' );  } else {  x = - x;  }  uart\_put\_int\_negative( x );  } |

Calculating the digits is simple: the rightmost digit is the value modulo 10 (the remainder of the division of the value by 10), in C: x % 10. For the next right-most-but-one digit we must first divide the value by 10, and then proceed as before. If the value-divided-by-10 is 0 we must stop. But the uart\_print\_int\_negative() function must also solve the problem of printing the digits in the reverse order in which they are calculated. This is done by printing the digit AFTER the recursive call to itself that is responsible for printing the higher digits.

|  |
| --- |
| void uart\_put\_int\_negative( int x ){  int xx = x / 10;  if( xx != 0 ){  uart\_put\_int\_negative( xx );  }  uart\_put\_char( '0' - ( x % 10 ));  } |

The uart\_put\_int\_hexadecimal() function is different from its decimal counterpart in that it handles (only) unsigned values, and the number of digits to be printed is specified by the caller. It is split in two functions, but for a different reason: printing one hexadecimal digit is sufficiently ‘complex’ to make it a separate function. It first mask of all bits except the lower four (a hexadecimal digit covers four bits). Next it checks whether the value is below 10. If is, we get the character to print by adding ‘0’. If not, we get it by subtracting 10 and adding ‘A’.

|  |
| --- |
| void uart\_put\_int\_hexadecimal\_digit( int x ){  x = x & 0x0F;  if( x < 10 ){  uart\_put\_char( x + '0' );  } else {  uart\_put\_char( ( x + 'A' ) - 10 );  }  } |

To print a specified number of hexadecimal digits of a value we must first print the leftmost one. To do this, we shift the value n – 1 digit values to the right, so it becomes the rightmost digit. The we can leave the printing to uart\_put\_int\_hexadecimal\_digit(). A hexdecimal digit occupies 4 bits, so we must shift ( n -1 ) \* 4 positions. We must do this for n, n-1, n-2, etc. down to 0.

|  |
| --- |
| void uart\_put\_int\_hexadecimal( unsigned int x, int n ){  while( n > 0 ){  uart\_put\_int\_hexadecimal\_digit( x >> (( n - 1 ) \* 4 ) );  n--;  }  } |

The example program in the sw-uart directory demonstrates the use of the various uart\_put\_... functions. Note that the uart functions use the timer, hence both timer\_init() and uart\_init() must be called first. This main() is a bit unusual for an embedded program because after printing what it wants to print it is done and returns. But to what? There is no command prompt or Operating System to return to. The development environment we use (bmptk) puts a busy loop after the call to main, so we can safely return from main and the chip will effectively stop. Other developments might behave differently, varying from restarting the program to crashing (doing all kind of weird things).

|  |
| --- |
| #include "LPC11xx.h"  #include "timer.h"  #include "sw-uart.h"    int main( void ){  timer\_init();  uart\_init();  uart\_put\_string( "Hello world, the answer is " );  uart\_put\_int\_decimal( 42 );  uart\_put\_string( ".\n" );    uart\_put\_string( "You are " );  uart\_put\_int\_hexadecimal( 0xDEADBEEF, 8 );  uart\_put\_string( ".\n" );    return 0;  } |

The sw-uart program uses both the sw-uart library files (which are in its directory) and the timer files (which are in the directory of the timer example). Hence the Makefile must mention both libraries. We could mention the .h and .c files explicitly, as we did in the timer example, but this time we use a shortcut: we mention the two as LIBRARIES, which implies that for each there is a .h and a .c[[2]](#footnote-2) file.

By default the make system expects to find all relevant files in the current directory. If it needs to search in other directories too we must mention these in the SEARCH line, so there we put the path of the timer directory, in this case relative to our directory. We could also have expressed it as ‘$(BMPTK)/examples/v1th04/timer’. [[3]](#footnote-3)

|  |
| --- |
| # Specify project files (other than $(PROJECT).cpp) (if any)  SOURCES := timer.c sw-uart.c  HEADERS := timer.h sw-uart.h  # Specify directories, other than the current, that contain sources  SEARCH := ../timer |

## Controlling a hobby servo

A (hobby) servo is an electromechanical component that is used for instance in a model plane to control the orientation of the wing flaps. It contains a motor, a few gear wheels, a position sensor, and a feedback circuit that controls the motor to keep the output axis of the servo at the commanded angle. The servo has three wires: for ground, power, and the control signal. Hobby servos are built for 5V power, but the version in the hardware set works reasonably well at 3.3V.

|  |  |  |
| --- | --- | --- |
|  | **Wire** | **Function** |
| Brown | Ground |
| Red | Power (3.3V) |
| Orange | Digital pulse |
|  |  |

Compared to other components a servo can suddenly draw large currents, so we need to include a capacitor, between the board and the servo, which acts as buffer[[4]](#footnote-4). This component is polarized: the white stripe must be connected to the ground.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Elco:**  **white bar = ground = negative** |

The control signal is a positive pulse, which must be repeated every 10 .. 30 ms (the repeat interval is not critical). The angle to which the servo must rotate is specified by the length of the pulse. Our servo expects a pulse of 0.5 … 2.5 ms, corresponding to an angle of 0 .. 180°.

|  |
| --- |
| http://bansky.net/blog_stuff/images/servo_pulse_width.png |

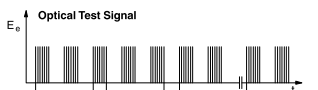
The demo code for the servo follows the pattern we have seen before: the init() function initializes the pin direction and sets the pin to the idle (low) level. The servo\_low() and servo\_high() functions make the pin low and high. These three functions are the only ones that ‘know’ to which pin the sevo is connected. The servo\_pulse() function sends one pulse to the servo of the length determined by its parameter, which must be expressed in microseconds.

|  |
| --- |
| #include "bmptk.h"  #include "timer.h"  void servo\_low( void ){  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ 0x10;  }  void servo\_high( void ){  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | 0x10;;  }  void servo\_init( void ){  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | 0x10;  servo\_low();  }  void servo\_pulse( unsigned int length ){  servo\_high();  await( now() + length );  servo\_low();  }  int main( void ){  unsigned int n;  servo\_init();  timer\_init();  for(;;){  for( n = 500; n < 2500; n = n + 10 ){  servo\_pulse( n );  await( now() + 25 \* 1000 );  }  }  } |

The main() slowly rotates the servo from 500 (0°) to 2500 (180°) and then back to 500 at once.

## IR remote control basics

An IR (Infra-Red) control as is used for TVs sends it commands using the IR light emitted by an IR LED. To prevent interference by ambient light, the receiver is encased in plastic that is ‘colored’ to pass only the IR light, and in addition to that the IR light is modulated (switched on an off) at a certain frequency. The receiver is optimized to receive only IR light variations of that frequency. The IR receiver we use is optimized for 36 kHz.[[5]](#footnote-5) On top of this, there receiver will ‘suppress’ any signal that is always present, hence it will only respond when we make the signal intermittent, for instance 1 ms on / 1 ms off. Hence the signal will look like this:



The receiver filters out the 36 kHz modulation, and its output is active low (high in rest, low when a signal is present). Hence the output for the above test signal will be:

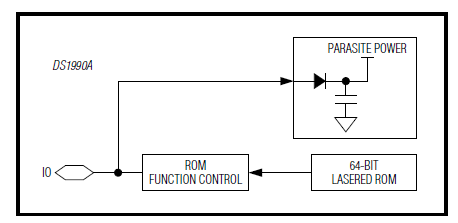


The IR receiver output can be connected to a GPIO pin of the LPC1114. To make the received signal visible a LED can be connected to the receivers output pin. Because the output pin is active low, the LED must be connected between the output pin and the power. The LED will be activated only part of the time, hence we use a 100 Ohm series resistor (brown-black-brown) instead of a 1kOhm version so the LED will be bright enough.

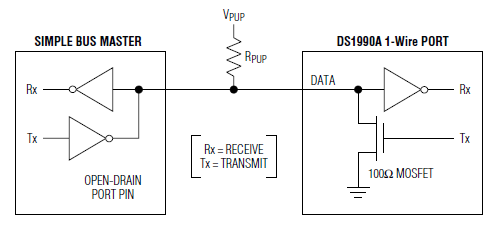
|  |  |
| --- | --- |
|  |  |

## The Dallas/Maxim one-wire bus

The Dallas/Maxim one-wire bus is a way to connect multiple peripheral chips to a microcontroller using only one wire (and ground of course, no circuit can be complete with only one wire) for communication in both directions. Additionally, that one wire can also be used to power the peripheral chip. To make this possible the chip has a diode and a capacitor, to ‘harvest’ a little energy when the wire is high, so it can ‘survive’ the short periods when the line is low.



The one line must have a pull-up resistor to make it high when in rest, and each device connected to the line (including the microcontroller) can pull the line low. All activity on the line is started by the microcontroller, who is the master of the line.



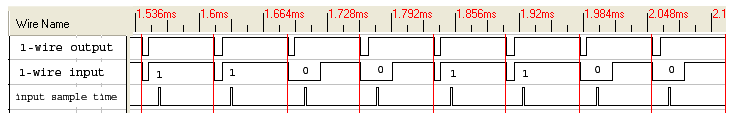
To initiate a data transfer, the master must pull the line low for at least 480 µs. After this reset pulse, the slave(s) respond by pulling the line low for a brief period. This presence pulse can be used by the master to detect whether any slaves are present on the line. A good moment for the master to sample the line is 30 µs after the end of the reset pulse.



Now the master can send a command byte over the bus. A 1 bit is sent by pulling the line low for 1..15 a 0 bit is sent by pulling the line low for 60..120 µs. Between the bits there must be some recovery time (the slave must recharge its power capacitor!), 10 µs will do. A byte is transmitted LSB-first. During this transmission there is no activity on the line from the slave.



If the command requires a response from the salve, the master can now pull the response bytes from the slave. The master does this by briefly pulling the line low (5 µs will work). To send a 1 bit back to the mater, the slave does nothing. To send a 0 bit, it pulls the line low for a short period. A good moment for the master to sample the line is 5 µs after the end of his ‘bit-request’ pulse. Like for the command byte, the response byte(s) are transmitted LSB-first.



The 1-wire device that we will use is an identity button. Such buttons are often used to identify a person. Our buttons are relatively simple: they contain a unique identification code that can be read out by the master. The code is also engraved on the button. For high-security situations more complex buttons are used, that implement a challenge response protocol.

|  |  |
| --- | --- |
| http://www.voti.nl/common/ds1990.jpg | [File:I-button.jpg](//upload.wikimedia.org/wikipedia/commons/5/56/I-button.jpg) |

The outer case of the button is the ground, the ‘text plate’ is the bus. We will use the GPIO pin 0\_4 (marked SCL on the board), which has a pull-up resistor on the DB103 board. The clothes peg in the set might be handy to make the connection to the button. We will use one command: 0x33, which asks the button to send its 8-byte identification code.

A skeleton application is provided, which you will have to complete. The only functions that known which pin is used are ds\_pin\_set() and ds\_pin\_get(). These must be implemented by calling the pin library that you have created. The ds\_pin\_pulse() generates a (low) pulse of the indicated duration. For its timing it uses the timer library you have created. Remember to include the relevant header files, and to change the Makefile to find and include your libraries.

|  |
| --- |
| / read a DS1990, use pin GP0\_4 ('SCL')  #include "bmptk.h"  #include . . .  void ds\_pin\_set( unsigned char x ){  if( x ){  . . .  } else {  . . .  }  }  unsigned char ds\_pin\_get(){  return . . . ;  }  unsigned char ds\_pin\_pulse( int t ){  ds\_pin\_set( 0 );  delay( t );  ds\_pin\_set( 1 );  } |

The ds\_present() function tests whether a device is present on the one-wire bus. It preforms the following tests:

* In rest, the line must be high
* 30 µs after the reset pulse, the line must be low (presence pulse)
* 1000 s after the presence pulse, the line must be high again

When a test fails, the function returns 0 (false), when all tests succeed, it returns 1 (true).

|  |
| --- |
| unsigned char ds\_present(){  // give the device time to get back to the 'rest' state  ds\_pin\_set( 1 );  delay( 2000 );    // test for stuck-at-0  if( ! ds\_pin\_get() ){ return 0; }    // request a presence pulse  ds\_pin\_pulse( 1000 );    // test for the presence pulse  delay( 30 );  if( ds\_pin\_get() ){ return 0; }    // test whether the presence pulse ends  delay( 1000 );  if( ! ds\_pin\_get() ){ return 0; }    // all tests succeeded  return 1;  } |

You must write the functions that send one bit and send one byte. You can use the UART code as inspiration for the send byte function.

|  |
| --- |
| void ds\_send\_bit( unsigned char b ){  . . .  }  void ds\_send\_byte( unsigned char b ){  . . .  } |

You must complete the receive bit function. Re-read what the master must do to request a bit from a slave.

|  |
| --- |
| unsigned char ds\_receive\_bit( void ){  unsigned char x;  . . .  x = ds\_pin\_get();  delay( 100 );  return x;  } |

The rest of the program is provided. The ds\_receive\_byte() function calls the the ds\_receive\_bit() function eight times, and assembles the bits into a byte value, which it returns. The bits are received starting with the lowest bit. The bits are inserted into the byte at the highest position, and shifted towards the lower positions before a new bit is inserted. Hence the first bit that was received ends up at the lowest position. To test your understanding of the process: What would happen if the shifting was done AFTER the new bit was inserted?

|  |
| --- |
| unsigned char ds\_receive\_byte(){  int i;  unsigned char d = 0;  for( i = 0; i < 8; i++ ){  d = d >> 1;  if( ds\_receive\_bit() ){  d = d | 0x80;  } else {  d = d & ~ 0x80;  }  }  return d;  } |

Note that the main() uses the UART, so you must include its header and instruct the Makefile to include its implementation. The main() checks each half-second whether a device is present on the one-wire bus. If so, it calls ds1990\_id\_print(), which sends the 0x33 ‘read ID’ command, receives the 8-byte ID code, and prints the two relevant bytes of it in hexadecimal format.

|  |
| --- |
| void ds1990\_id\_print(){  int i, d;  ds\_send\_byte( 0x33 );  for( i = 0; i < 8; i++ ){  d = ds\_receive\_byte();  uart\_put\_int\_hexadecimal( d, 2 );  uart\_put\_string( " " );  }  uart\_put\_string( "\n" );  }    int main( void ){  timer\_init();  uart\_init();  uart\_put\_string( "DS1990 reader\n" );  for(;;){  delay( 500 \* 1000 );  if( ds\_present() ){  uart\_put\_string( "\*\*\*\n" );  ds1990\_id\_print();  } else {  uart\_put\_string( "-\n" );  }  }  return 0;  } |

# Week 3 – analog to digital (A/D) and back (PWM)

## Changing the configuration of a pin

We have used GPIO pins of the LPC1114 to read digital values (0 for low, 1 for high). Some GPIO pins have other functions, for instance as Analog-to-Digital (A/D) inputs. Each pin has an IOCONFIG register which controls the properties of the pin. The description of these registers shows the interpretation of the bits and the default (reset) value. Most (but not all!) pins default to digital I/O. The IO configuration block is one of the parts of the LPC1114 that is disabled by default. Hence when we want to change the configuration of a pin we must first enable the clock to the IO configuration by writing a 1 to the appropriate bit.

|  |  |
| --- | --- |
|  |  |

One interesting pin is GPIO 0\_0. By default this is the reset pin of the chip, when we make this pin low the chip will reset (starts its program from the beginning). For this purpose, the DB103 board has a reset switch (next to the DB100 connector). If we want to use this pin as normal input (or output) we must change the appropriate bits in the configuration register from 000 to 001[[6]](#footnote-6). As always, take care not to change any other bits!

|  |
| --- |
|  |
|  |

The example program reset-as-input demonstrates the use of the reset pin as normal input. It is almost the same as the read switch program we saw last week.

|  |
| --- |
| #include "bmptk.h"  void led\_init(){  LPC\_GPIO1->DIR = LPC\_GPIO1->DIR | ( 1 << 4 );  }  void led\_set( unsigned char x ){  if( x ){  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA | ( 1 << 4 );  } else {  LPC\_GPIO1->DATA = LPC\_GPIO1->DATA & ~ ( 1 << 4 );  }  }  void switch\_init( void ){  . . .  }  unsigned char switch\_get(){  return ( LPC\_GPIO0->DATA & ( 1 << 0 ) ) == 0;  }    int main( void ){  led\_init();  switch\_init();  for(;;){  led\_set( switch\_get() );  }  } |

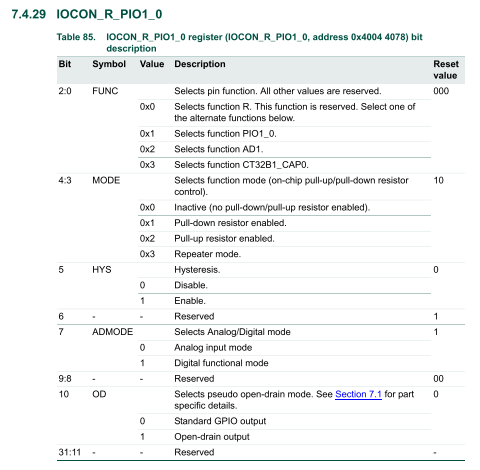
The only special part is the switch\_init() function. This function first saves the value of the clock control word, and sets the bit in this word that enables the clock to the IO configuration. At the end, it restores the old value of the clock control word. This is not strictly required, but it is a good principle for a function to leave things as they were before it was called (except of course for the things it MUST change).

To set the lowest three bits of the LPC\_IOCON->RESET\_PIO0\_0 to 001 we should not assume what the previous value was. Hence we first clear the three bits by ANDing with a value that is all 1’s except for the lower three bits that are 0. Next we OR with the value 1, which sets the lowest bit. The end result is that we only changed the three lowest bits, and that their new value is 001b.

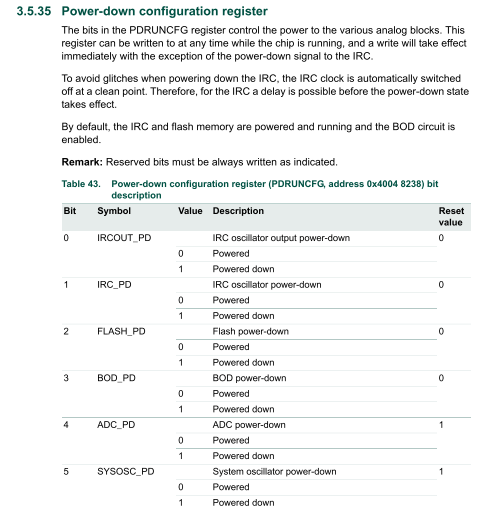
|  |
| --- |
| void switch\_init( void ){  unsigned int save\_clockcontrol = LPC\_SYSCON->SYSAHBCLKCTRL;  LPC\_SYSCON->SYSAHBCLKCTRL |= (1<<16);  LPC\_IOCON->RESET\_PIO0\_0 &= ~0x07;  LPC\_IOCON->RESET\_PIO0\_0 |= 0x01;  LPC\_SYSCON->SYSAHBCLKCTRL = save\_clockcontrol;  } |

## Analog to digital conversion

The LPC1114 chip has an Analog-to-Digital (A/D) conversion circuit, which can convert an analog value to a 10-bit digital value. In the most common configuration the digital value 0 corresponds to 0 Volt, and the digital value 0x3FF (1023 decimal) corresponds to the power supply of the chip, in our case 3.3 Volt. This A/D converter can be ‘connected’ to a subset of the GPIO pins. The configuration register of GPIO 1\_0 shows that AD1 (analog input 1) is one of the alternate functions of this pin. To select this function we must write the value 0x2 in the lower three bits of the pin configuration register, and bit 7 (ADMODE) must be set to 0. Note that to do this, we must first enable the IO configuration block by setting the appropriate bit is SYAHBCLCKCNTRL.

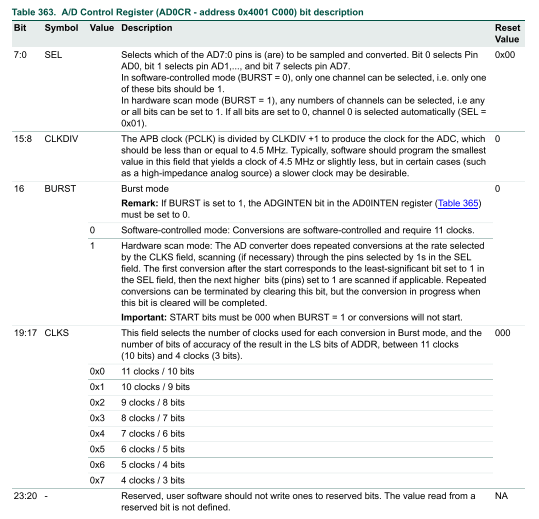


The A/D converter contains analog circuits. Unlike digital circuits, analog circuits can’t be powered down by just removing the clock signal, so the chip by default switches off the power to such circuits. To use the A/D converter we must enable its power by clearing the correct bit in the PDRUNCFG register.

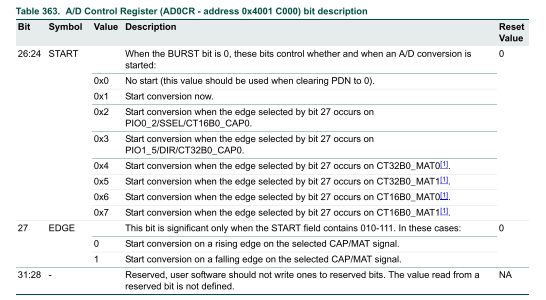


The next step is to configure the A/D converter itself. This involves four aspects:

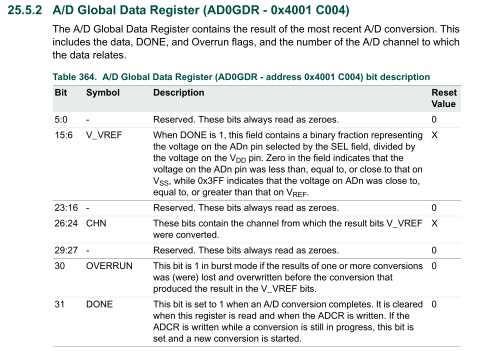
* In the lower 8 bits we must specify which of the AD0..AD7 pins we want to use. Of these 8 bits the bit corresponding to the pin must be set (1), the other bits must be cleared (0).
* The next 8 bits specify the factor by which the processor clock frequency (12 MHz) is to be divided to produce the A/D conversion clock (which must be 4.5 MHz or lower).
* Bit 16 determines whether the A/D converter will perform a single conversion (software-control), or will scan all pins for which a bit in the SEL field is set (hardware scan). We use the single conversion mode.
* The bits 17-19 determine how many significant bits the A/D converter will produce. We want 10 bits.



When the configuration work is done we can start the A/D converter by setting the START field to 0x1. Note that this can’t be done in the same instruction that changed other bits of the configuration register.



The A/D converter takes some time to complete the conversion, so we must now wait until the converter hardware indicates that the result is ready. It does so by setting the DONE bit in its global data register. Note that you can read this 1 only once: it is cleared when the register is read. The A/D conversion result is available in bits 6..15 of the global data register.



The full names of the registers involved in the A/D conversion are show in the table below.

|  |  |
| --- | --- |
| **Register name** | **use** |
| LPC\_SYSCON->SYSAHBCLKCTRL | Enable the clock to the pin configuration and A/D converter. |
| LPC\_IOCON->R\_PIO1\_0 | Configure the pin for analog input. |
| LPC\_SYSCON->PDRUNCFG | Enable power to the A/D converter. |
| LPC\_ADC->CR | Configure the A/D converter. |
| LPC\_ADC->GDR | Start the A/D converter and get the conversion result. |

The set contains a Light Dependent Resistor (LDR). Its resistance gets lower when more light falls on its surface. To use the LDR to get a reading of the amount of light we must use it with another resistor (10kΩ, brown-black-orange) to present to the A/D input pin a fraction of the power voltage. This fraction will be lower when more light falls on the LDR. And LDR is a bit slow, you should wait ~ 300 ms to get a stable reading.

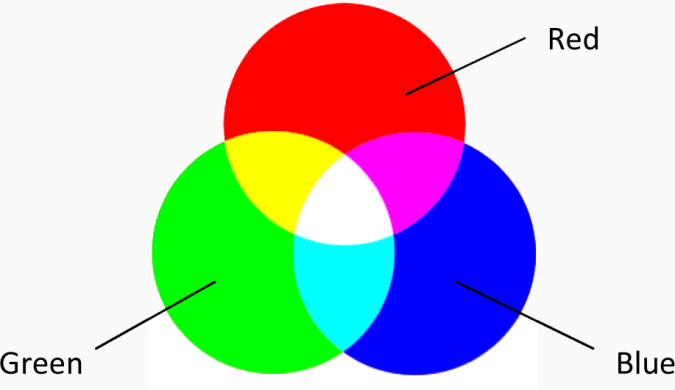
|  |  |
| --- | --- |
|  |  |

Another analog sensor in the kit is the LM335 temperature sensor. It is used the same way as the LDR, but with a 1kΩ (brown-black-red) pull up. The LM335 will keep the voltage across the chip to the temperature in degrees Kelvin \* 10 mV. Check the datasheet of the LM335 for the pins. Note that you must leave one pin unconnected.

|  |  |
| --- | --- |
|  |  |

## A color sensor

Color as we see it with the color-sensitive cone cells in our retina is composed of three values: the red, green, and blue intensity. When white light hits a surface and we see the surface as green, this is because the surface absorbs the red and blue parts of the white light, and reflects only the green part. When the surface absorbs only the blue part it will reflect the red and green parts, which we will perceive as yellow.



A color sensor determines the color of an object in front of it. One way would be to shine white light on it and determine which color(s) of light it reflects, but that requires the use of either sensors that are sensitive to a specific color, or a set of filters that pass only a specific color. Instead we use three LEDs to illuminate the surface with a specific color. When the LED color is not absorbed by the surface we expect most light to be reflected, and hence to have effect on our LDR. When the light is absorbed we expect little effect on the LDR. For this to work we must make sure that the LEDs do not shine directly on the LDR.

The next photos show how you can build a color sensor for the pieces of paper that are part of the set. The piece of black tube is cut and used to shield the LEDs and the LDR from each other. Note that these tubes must be somewhat shorter than the outer shield (around the total construction) so each LED can illuminate the part of the paper above the LDR.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

The blue LED is much more effective than the red and green LEDs, hence we use a 1kΩ (brown-black-red) for the blue LED, and 100Ω resistors (brown-black-brown) for the red and green LEDs.

To get a good result, we must take a number of steps. The values used in these steps depend somewhat on the construction of the sensor and the ambient light, so when you have prepared your sensor at home you might need to repeat some steps before demonstrating in another environment.

We are interested in the effect of shining colored light on our test surface; hence we must first measure the LDRs A/D reading without any light (dark, all LEDs off). Next we take three readings with a white piece of paper, one with the red LED on, once with the green, and one with the red. These three values will be lower than the dark reading. Calculate the three differences.

Ideally the three values we have now should be equal, but due to the different efficiencies of the LEDs and the properties of the LDR (which is not equally sensitive to all colors) the three values will be different. So the next step is to scale the three values: multiply them by the largest of the three, then divide each by their original value. Now you have three values that – for white paper – are roughly the same. Without any paper on top of our sensor the scaled values should be small (compared to the white-paper values).

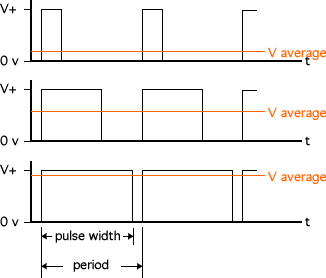
Now we can test the effects with colored pieces of paper. The scaled values should be sufficient to distinguish between white, blue, green and red paper, and no paper at all.

## Pulse Width Modulation

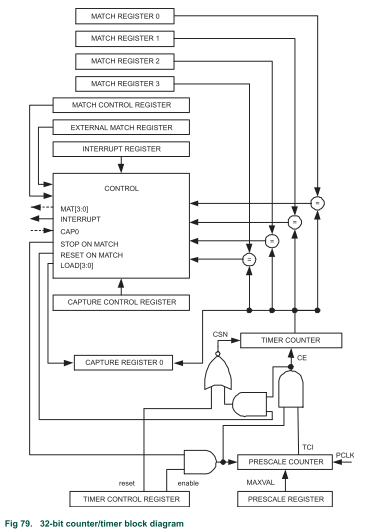
One way to regulate the amount of electrical power to a device is to put a resistor in series with it. The downside of this approach is that energy will be wasted in this resistor. For high-power device this will be a lot of energy. This is a waste in itself, and means that the resistor must be large, expensive, and will get hot. It is not a coincidence that the rheostat below, which was meant to dim for instance a theatre light, resembles a small heater.

[](http://img.directindustry.com/images_di/photo-g/power-rheostats-64090-2859617.jpg)

A better way to regulate the power to a device is to use Pulse Width Modulation. In essence this switches the full power to the device on and off at a high frequency. The ratio between the on and off times determines how much power the device gets. The frequency must be chosen high enough that the device (and people) doesn’t ‘notice ‘ it, yet low enough that the effects of switching (which always causes some loss) don’t dominate.



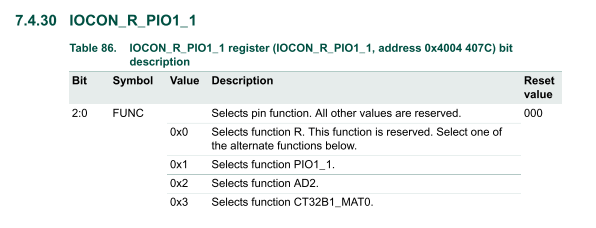
A PWM signal can be created in software, but that takes a lot of attention from the CPU, and can be difficult for the programmer to integrate with the rest of the software. Alternatively, most microcontrollers have built-in hardware that can generate a PWM signal. In our LPC1114 this hardware is part of the timers. Each timer has four match registers, which trigger when the value in the counter is equal to the value in the match register. Another register specifies the action(s) that will happen when such a match occurs. Possible actions include the setting or resetting of an IO pin, and the resetting of the counter register. A combination of these actions can be used to generate a PWM signal.



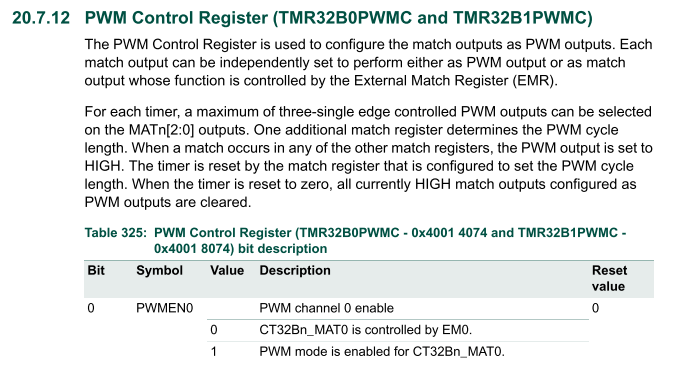
We already use TMR0 for timing, so we will use TMR1 to generate a PWM signal. To do this, we must take a number of steps that are familiar from the use of TMR0:

* We must enable the clock to TMR1 (note that TMR0 and TMR1 have separate clock enable bits in the SYSAHBCLCKCNTRL register);
* We must set the Prescale Register to an appropriate value;
* We must set the Counter Control Register to select the appropriate clock source;
* We must start the timer (in the Timer Control Register).

To get a PWM signal we must take some additional steps. First we must configure a pin to be a PWM pin instead of a GPIO. We will use pin 1\_1, which can, according to its configuration register, be CT32B1\_MAT0, which means ‘MATch register 0 of the 32 bit Counter Timer 1’.



Next we must enable PWM mode for match channel in the PWM Control register.



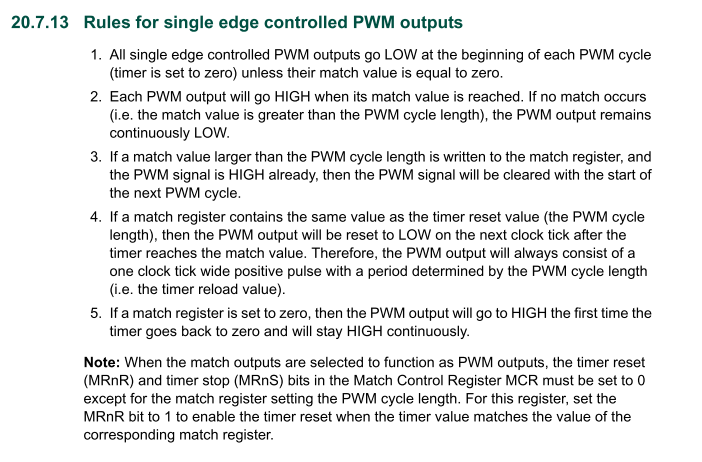
We will use two match registers, MR0 and MR3. MR3 will be used to reset the timer, hence it determines (together with the clock source and the prescaler) how long the PWM period is. This resetting must be enabled in the Match Control Register. Take care not to set any other bits. Resetting the timer counter will implicitly clear all pins that are configured as PWM for this timer.

|  |
| --- |
|  |
|  |

The pin is now cleared when the timer is reset, and the timer counts up to the value in MR3. The last step is to have the pin set when the counter matches MR0. This is configured in External Match Register.

|  |
| --- |
|  |
|  |

What we have configured is that an MR0 match will set the output pin high, after which an MR3 match will clear the timer and set the output pin high. Hence the pin will be low for ( MR0 / MR3 ) of the time.



The next table shows the resisters involved in configuring the PWM output.

|  |  |
| --- | --- |
| **Register name** | **use** |
| LPC\_SYSCON->SYSAHBCLKCTRL | Enable the clock to the pin configuration and TMR1. |
| LPC\_IOCON->R\_PIO1\_1 | Configure the pin for CT32B1\_MAT0 output. |
| LPC\_TMR32B1->CTCR | Select the cock source. |
| LPC\_TMR32B1->PR | Configure the prescale factor for the timer. |
| LPC\_TMR32B1->TCR | Start the timer. |
| LPC\_TMR32B1->PWMC | Configure CT32B1\_MAT0 as PWM (hence it wll be celared when the counter is reset). |
| LPC\_TMR32B1->MCR | Configure an MR3 match to reset the counter to 0. |
| LPC\_TMR32B1->MR3 | Maximum counter value. |
| LPC\_TMR32B1->EMR | Configure MR0 to set the CT32B1\_MAT0. |
| LPC\_TMR32B1->MR0 | Match value at which the CT32B1\_MAT0 will be set. |

# Week 4 – clocked communication: SPI, I2C

## Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is a high-speed interface that is often used between a controller and its peripheral chips. The hardware involved is rather simple: the peripheral chip (slave) contains one or more shift registers. The controller (master) drives the enable, clock and serial-data-in lines of these shift registers, and (for data from the slave to the master) it receives the serial-data-out. When data is shifted into a shift register its outputs show the data as it is being shifted in. When this data has a direct effect, for instance because it is interpreted as a command, these intermediate values can produce undesirable effects.

|  |  |
| --- | --- |
|  |  |

The 74HC595 is often used as a cheap and simple way to give a microcontroller more output pins. It requires 3 microcontroller pins to drive this chip, which provides 8 output pins. To prevent intermediate values from being visible at the output pins the 74HC595 has a second register, the holding register, which is loaded with the output of the shift register when the storage clock is pulsed.

|  |  |  |
| --- | --- | --- |
|  | Pin1 |  |

The 74HC595 was invented long before the SPI protocol was described, hence the names pins of this chip do not match the SPI names. To make things even worse, the 595 is a ‘generic’ chip, produced by different manufacturers, who each have their slightly different version of the pins names. The table below shows the correspondence. Note that the 595 chip has no pin that corresponds to the MISO pin, because it is an output-only chip.

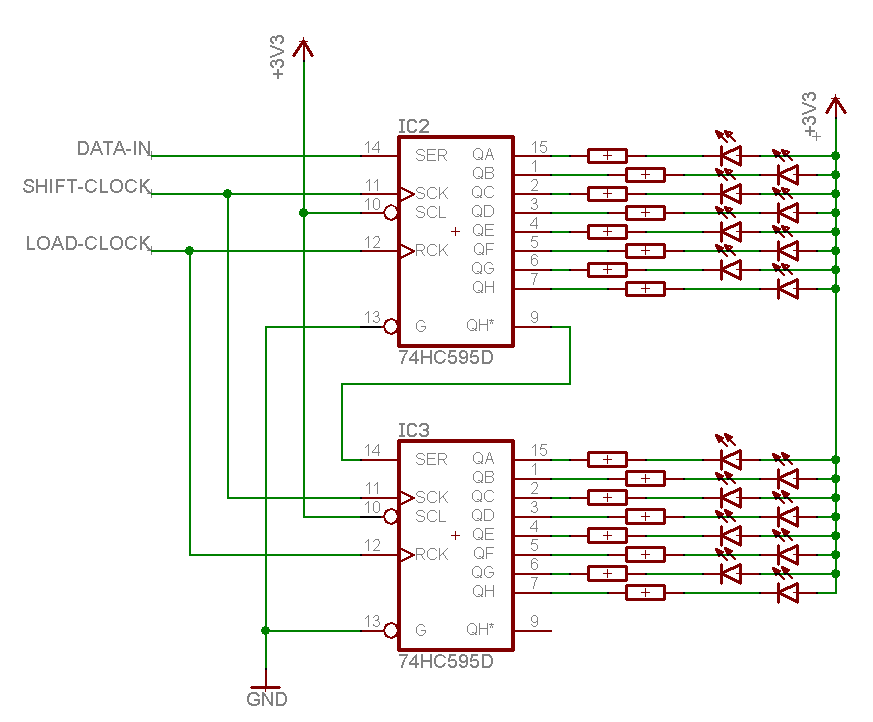
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SPI name and function** | | **74HC595 names** | | | |
| SCLK | System CLocK | SHCP | Shift Clock Pulse | SCK | Shift CloK |
| MOSI | Master Out Slave In | DS | Data Signal | SER | Serial Data |
| MISO | Master In Save Out | - | | | |
| SS | Slave Select | STCP | Store Clock Pulse | RCK | Register CloK |

The next figures are copied from the 74HC595 datasheet. They show the effect of the various pins. The rightmost picture shows the internal organization of the chip, with the chain of 8 registers that form the shift register, which each output to a register that is part of the 8-bit holding register.

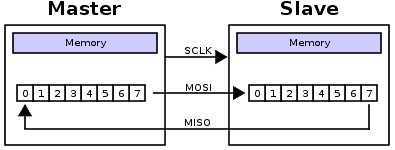
|  |  |
| --- | --- |
|  |  |

The 74HC595 provides the output of the 8th shift register on an external pin, which can be used to daisy-chain a next 74HC595 register. Hence 3 output pins of the microcontroller can control a virtually unlimited number of 74HC595 output pins. Next diagram shows how to connect a chain of two chips. It also shows what to do with the two extra pins that are no use to us:

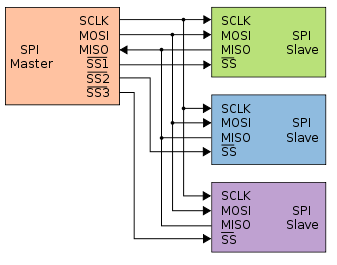
* the output enable or G (gate) pin must be connected to ground to enable the outputs;
* the Master Reset or SCL pin must be connected to the power to keep the chip out of reset.



A full SPI interface transfers data from the master to the slave (like with a 74HC595), and simultaneously transfers (on the same clock pulses) data from the slave to the master. The next picture suggests that this data originates from the same shift register in the slave that accepts the data from the master, but this need not be the case.



Multiple slave chips can share the SCLK, MOSI and MISO pins of the master, but they must each have an individual Slave Select (SS) pin. When a slave is not selected, it will not respond to the data clocked out by the master, and it will not drive its MISO pin.



## Inter-IC communication (I2C)

The I2C or Inter-IC Communication bus is another protocol for communication between a microcontroller and its peripheral chips. It differs from SPI in a number of aspects:

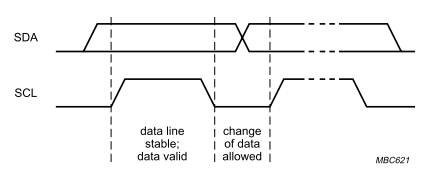
* Designed for lower speed and (consequently) longer distances (although typical use is, like SPI, within one PCB);
* Uses only 2 wires, compared to (3 + 1 per slave) for SPI;
* More complex protocol requires more logic in the slave and more CPU attention in the master.

I2C uses two lines, called SDA (Serial Data) and SCL (Serial CLOck), which both have a pull-up-resistor. All chips connected to the I2C bus can pull the lines low, but no chip can pull the line high. This arrangement is called an open-collector (or open-drain) bus, or sometimes a wired-OR bus.

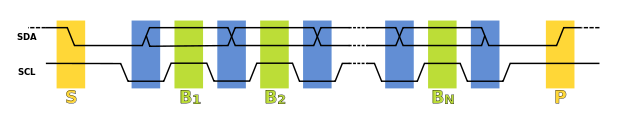
|  |  |
| --- | --- |
|  |  |

The I2C bus was invented by Philips, who had a patent on it. Hence other manufacturers that implemented the I2C bus on their chips had either to pay royalties to Philips, or tried to avoid this by implementing a protocol that was compatible with I2C, without mentioning I2C. The I2C patent has expired, but you can still find many chips that are described as ‘implementing a two-wire protocol’ or something similar. In most cases this means that they implement I2C.

At the bit level, the I2C bus uses one clock pulse (on the SCL line) to transfer one bit (on the SDA line). The protocol requires that while transferring bits, the SDA line must be stable while the SCL line transitions or is high. The bits of a byte are transferred MSB (most significant bit) first. The SCL line is always driven by the master, the SDA line is driven by the device on the bus that sends the bit[[7]](#footnote-7).



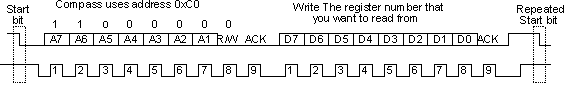
Transitions of the SDA line while the SCL line is higher are used to signal two special conditions: the S (start) condition (SDA goes low while SCL is high) is used to mark the start of an exchange. You can see it as a sort of reset signal. The P (stop) condition is used to mark the start of an exchange.



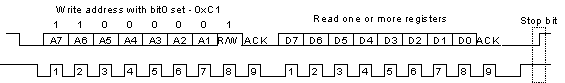
The first byte of an exchange is the command byte, and it is always transmitted from the master to the slaves. As all bytes, it is 8 bit long and is followed by an acknowledge bit that is sent by the slave that is addressed. This bit must be low to let the master know that the slave is present. The command byte consists of 7 address bits that identify the slave, and a last bit (the least significant bit) that indicates whether the exchange will be a write (from master to slave) or a read (from slave to master). When the bit is low (0) it indicates a write, a high bit (1) indicates a read.

|  |  |
| --- | --- |
|  |  |

The next diagram shows a one-byte write to a slave chip at chip address 1100000b. The 8th bit is low, indicating a write. The 9th bit is driven low by the slave. The next 8 bits are again driven by the master, then the slave drives the acknowledge bit for this second byte. Next the master could have sent a stop bit, but instead (as a shortcut that is allowed by the I2C standard) its sends a start bit, which both terminates the previous exchange and starts a new one.



To read data from a slave the master must use a read command. After the command byte, the slave will send data bytes, and now the master must send the acknowledge bit after each byte. Note that there is no way for the master to include some address within the slave in a read exchange. The custom is that such an address is transferred to the slave by a previous write exchange. The next read operation shows one byte being read, after that the master terminates the exchange with a stop bit. But the master could have read any number of bytes (up to what the slave chip permits).



Like with SPI, the I2C standard does not state what the data bytes that are transferred mean. This is described in the datasheet of the specific chip.

The I2C specification describes a number of variations that we can ignore because the chip that we will use does not implement or require these variations:

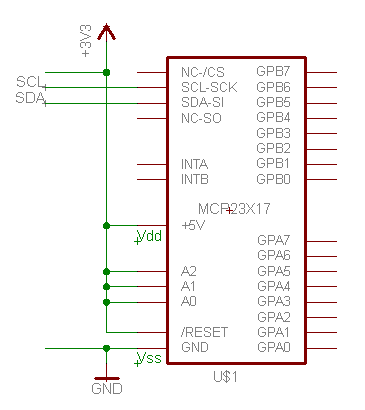
* Repeated start condition: terminate each exchange with a stop condition;
* 10-bit addressing: used to get a larger chip address space than the 128 addresses allowed by a 7-bit chip address;
* Clock stretching by the slave: this is rarely used, and our slave chip doesn’t use it;
* Multi-master arbitration: an I2C bus can have multiple master, in which case a protocol must use to avoid that two masters start an exchange at the same moment;
* Higher speed I2C (400 kHz and 3.4 MHz): the chip we use supports 400 kHz, but we can use any lower frequency we want.

## The MCP23017 I2C I/O extender

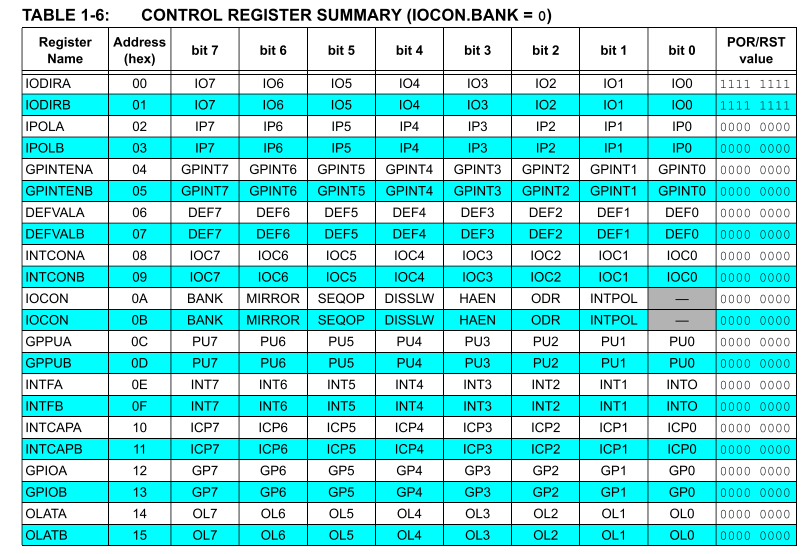
The MCP23017 I/O extender chip provides access to 16 I/O pins via an I2C bus. These are real I/O pins: they can be configured as input or as output, and like the LPC1114 I/O pins each pin has a weak pull-up resistor that can be enabled or disabled under software control. (At start-up, these pull-ups are disabled.) Additionally the chip has two interrupt outputs that we won’t use, and a rest pin that must be tied high to keep the chip active.

|  |  |
| --- | --- |
|  |  |

As with most I2C chips, 4 of the address bits are fixed, and the remaining 3 can be configured by 3 inputs to the chip. In the circuit diagram below they are all tied high, so the address of the chip is 0100111b.



The MCP2017 contains 16 registers of 8 bits each. These registers can be addressed in two ways, depending on a bit in the IOCON register. The next table shows the default addressing. For each of the two 8-pin ports, there are 3 registers that we must use. The IODIR registers determine, for each pin, whether it is an input (the IODIR bit is 1) or output pin (the IODIR bit is 0). As the table shows, all bits default to input at startup. The GPPU registers enable or disable the pull-up resistors, a 1 bit enables a pull-up resistor. The GPIO registers are ‘connected’ to the I/O pins. For an output pin a bit written to a GPIO register will set the pin according to that bit. For an input pin a read from the GPIO register will reflect the level on that pin.



The MCP23017 supports a number of I2C transactions, as shown in the next diagram. We will use only 3:

* In a byte write transaction the master first sends the command byte, then the address (within the chip) to which the byte must be written, and finally the data byte itself.
* A read operation requires that we first set the address from which we want to read. This is done by a truncated write transaction: after the command byte and the data byte the master sends a stop (P) instead of a data byte.
* When the address has been set, a read transaction starts with the master sending a command byte (with the R/W bit set to read), after which the slave sends a data byte.

All transactions are terminated by the master by sending a P (stop) condition.

|  |
| --- |
|  |

An I2C low-level library is provided that implements the operations shown below. You will have to implement the higher-level I2C and MCP23017-specific operations using these lower-level operations.

|  |
| --- |
|  |

# Week 5 – multiplexing

## Driving a multiplexed seven-segment display

A seven-segment display consists of 8 LEDs (7 for the segments, one for the decimal point) arranged to form a digit. To save pins, one side of each LED is connected to a common pin, the other side has an individual pin. There are two flavors: common cathode (negative side is common) and common anode (positive side is common).

|  |  |
| --- | --- |
|  |  |

The segments are identified by a letter A..F. It is common practice to connect segment A to bit 0, B to bit 1, …. , and the Decimal Point to bit 7, as shown in the table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | Bit | Display segment | | 0 | A | | 1 | B | | 2 | C | | 3 | D | | 4 | E | | 5 | F | | 6 | G | | 7 | DP | |

For a multiple-digit display the segments of each digit are connected, again to save pins. The figure shows the pins of the displays that we use (either one 4-digit display, or two 2-digit displays). For the 2-digit displays you must connect the segment pins of the two displays together (pins 7 8 3 2 1 10 4 5).

|  |  |
| --- | --- |
|  |  |

10 9 8 7 6

|  |  |
| --- | --- |
|  | [LED-Anzeige LITEON LTD-2601G, 2 Digit, grün](http://cdn.pollin.de/article/xtrabig/X121143.JPG)  1 2 3 4 5 |

Such a display can’t simultaneously show a number on digit one and a different number of digit two, because each segment that has its negative side connected to ground will light up in all digits that have their positive side connected to the power. The digits are no longer independent.

The trick used to drive such a display is (time-division-) multiplexing. First only digit one is activated (its positive pin connected to the power), the other digits are deactivated (by either connecting their positive pins to ground or leaving them open). The segments that we want to light up for the first digit are now connected to ground via an appropriate resistor; the ones that we don’t want to light up are connected to power (or left open). This situation is left in place for some time. Then the next digit and its segments are activated for some time, etc. When we switch between the digits fast enough the slowness of the human eye will create the illusion that all four digits are shown. Two points deserve attention:

1. The amount of times spent on each digit determines how bright it will appear. Hence we must wait some time while a digit lights up (switching at the highest speed will not work well).
2. When switching from one digit to the next we must switch off the previous digit before activating the next one. If this is not done carefully there will be ghosting: the pervious digit will appear (very dimly) overlaid on the next digit.

There are two board ways to integrate the switching with other activities:

1. Cycle through all 4 digits, and then do the other activities.
2. Show one digit, do other activities, maybe wait some more, then switch to the next digit.

|  |  |
| --- | --- |
|  |  |

To display a value, you must convert the value to the bit pattern corresponding to the segments that must be lighted to show the value. This can be done with a table of constants, or with a chain of if statements. Either way, you will have to find the bit patterns that correspond to each value.

|  |  |
| --- | --- |
| const unsigned char  seven\_segment\_patterns[ 10 ] = {  0x45,  0x66,  . . .  0x1B  }  unsigned char pattern\_from\_value(  unsigned char x  ){  if( x >= 10 ){  return 0;  }  return  seven\_segment\_patterns[ x ];  } | unsigned char pattern\_from\_value(  unsigned char x  ){  if( x == 0 ){  return 0x45;  } else if( x == 1 ){  return 0x66;  . . .  } else ( if x == 0x1B ){  return 0x1B.  } else {  return 0;  }  } |

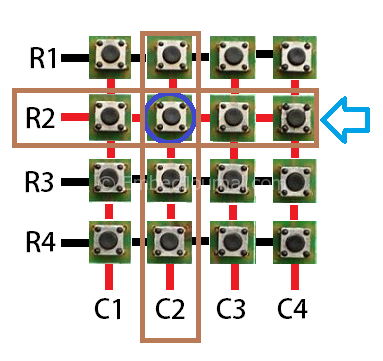
## Reading a matrix keypad

When a large number of switches must be interfaced to a microcontroller it is a waste of pins to connect each pin to one switch. Instead the switches are arranged in a matrix, at the cross points between vertical and horizontal wires. Those wires are connected to GPIO pins. This way N + M GPIO pins are connected N \* M switches.

|  |
| --- |
|  |

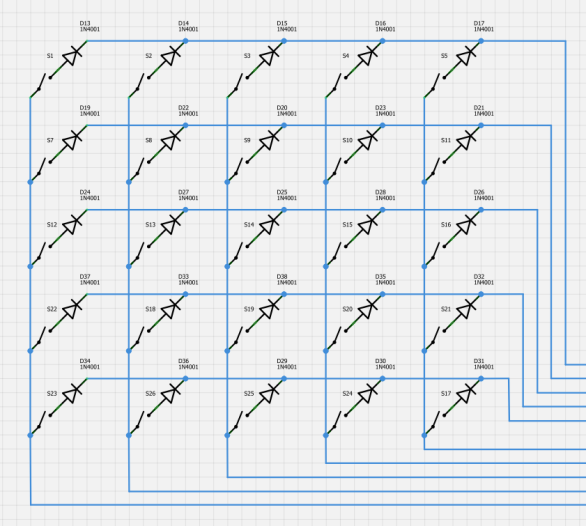
The trick to read an individual switch is

* One set of lines (let’s assume the rows) are inputs to the microcontroller, with pull up-resistors, so without any connection they will read as 1 (high)
* To read a switch its column (in the next figure C2) is made low, the other columns are left unconnected (in the microcontroller this is done by making these lines inputs).
* Now we read the row of the switch (in the next figure R2). When the switch is not pressed it will be 1 (due to the pull-up), when it is pressed it will be low (due to the connection via the switch to the column C2, which is low).



In fact we read all four switches on the column C2 in one go. Next we make another column low (for instance C3, and C1, C2 and C4 unconnected/input), and we can read the next column of four switches.

Note that this process assumes that only one switch is pressed at a time. When in our example for instance R3-C2 and R3-C3 and R2-C3 are pressed there is no way to detect whether R2-C3 is pressed, because the connection that it would make is already made by the other 3 switches. If it must be possible to read more than one key pressed at the same time, a diode must be put in series with each switch.



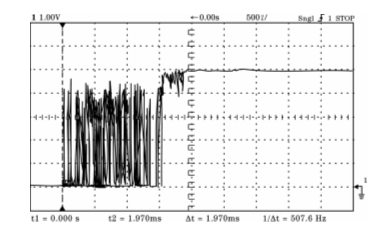
Another issue that we must address is that the weak pull-up inside the LPC1114 takes some time to pull a line high. A lower valued resistor would reduce this time, but why add extra hardware when the problem can be solved in software! The solution is to wait some time (5 ms is enough) before reading.

The skeleton code for reading one key from a matrix keypad is shown below. The row an column number of the key are passed as parameters. The read\_one\_key() function

* first initializes the pins as GPIO;
* makes them all inputs;
* makes one pin output and low;
* waits 5 ms;
* checks whether the one input is low, and if so returns 1;
* (when the pin was not low) returns 0.

|  |
| --- |
| char read\_one\_key( int column, int row )}{  // configure the pins as GPIO  // make them all inputs  // make the one column output and low  // wait 5 ms  // check whether the one row is low  // if so return 1  return 0;  }  char keypad\_read( void ){  if( read\_one\_key( . . ., . . . )){ return ’1’; }  . . .  return 0;  } |

A final issue with reading switches in general is that a switch that is closed literally bounces a few times, so the connection is made, broken, made again, broken again, etc. This happens fast: worst case the bounce will be over in 50 ms. The figure shows a real output of a switch.

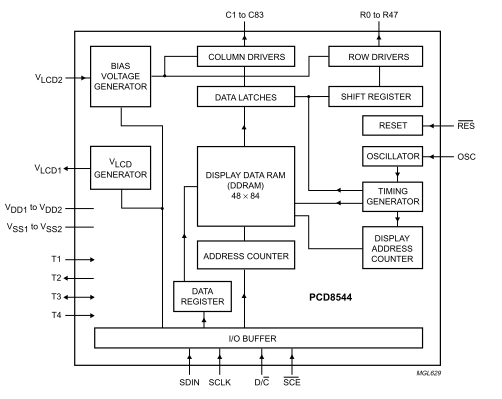


For some functions (like lighting a LED) bounce is not important, but if we want to treat the pressing of a switch as an even (for instance when we count the number of times the switch is pressed) we must not be fooled by the bouncing. The easiest way to handle key bouncing is to check the key no faster than 50 ms. This way we can have at most one check inside the bounce interval. If the key was low before and we read high in the bounce interval all is well. If the in the bounce interval we read the key low we detect the key press on the next read (maximum 50 ms later).

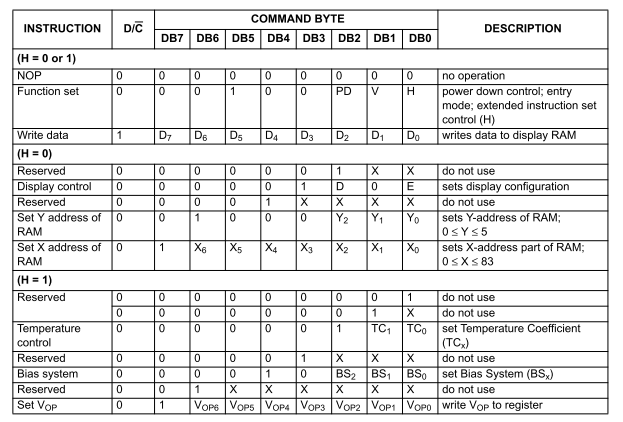
# Week 6 – graphics, and a small brother

## Using a Nokia 5510 graphics LCD

The Nokia 5510 is a small black-and-white 84 (horizontal) x 48 (vertical) pixel graphics LCD. It was used in various Nokia telephones and is used a lot by hobbyist because it is cheap and relatively easy to interface. It uses a PCD8544 LCD controller or some mostly-compatible clone of this chip. This chip has a SPI-like interface: in addition to the normal SCLK, SDIN (MOSI) and SCE (SS) pins it has a D/C pin that determines whether a byte that is clocked in is handled as pixel data or as command.



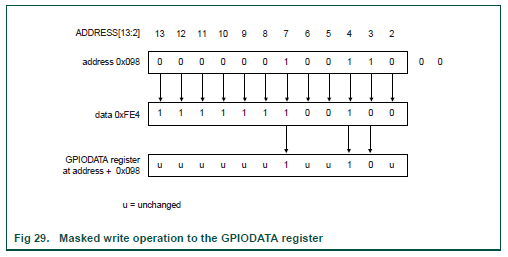
The chip must be configured in accordance with the properties of the LCD screen it is used with. This involves a number of parameters that are, to put it mildly, not very well documented. For the clone variants of the chip there some additional commands are required that are not found in the normal documentation.



A library is available for the LCD that has a simple interface: it can initialize the LCD, clear it, and draw a single pixel. Additionally there is a function that draws a line. This function is not LCD-specific: it uses the draw-pixel function. The details like the GPIO pins that are used to interface to the LCD can be found in the (Doxygen!) comments.

|  |
| --- |
| void lcd5510\_init( void );    void lcd5510\_clear( void ):  void lcd5510\_write( unsigned int x, unsigned int y, unsigned char d );  void lcd5510\_line(  unsigned int x0, unsigned int y0,  unsigned int x1, unsigned int y1,  unsigned char d ); |

The LCD library uses bit-banding to access the GPIO pins. This is a feature of the Cortex M0 that allows a bit (or set of bits) within a word to be used in isolation. It dedicates a range of addresses to accessing the bits within the word. The bits are selected by the lower bits of the address. As you can imagine this technique is very wasteful of addresses, hence it is used only for a few registers for which it makes sense.



|  |
| --- |
| void sce\_set( unsigned char x ){  if( x ){  \*(volatile int \*)( 0x50000000 + 1 \* 0x10000 + ( 0x04 << 0 )) = -1;  } else {  \*(volatile int \*)( 0x50000000 + 1 \* 0x10000 + ( 0x04 << 0 )) = 0;  }  }  void res\_set( unsigned char x ){  if( x ){  \*(volatile int \*)( 0x50000000 + 1 \* 0x10000 + ( 0x04 << 1 )) = -1;  } else {  \*(volatile int \*)( 0x50000000 + 1 \* 0x10000 + ( 0x04 << 1 )) = 0;  }  } |

As said, the interface to the PCD8544 is SPI, plus a command/data select line. The code reflects this: the functions lcd5510\_command() and lcd5510\_data() send one command or data byte to the controller chip. They do this by setting the command/data line, and then delegating to the lcd5510\_send\_byte() function, which clocks out the 8 bits.

|  |  |
| --- | --- |
| void lcd5510\_send\_byte(  unsigned char d  ){  int i;  for( i = 0; i < 8; i++ ){  sdin\_set( d & 0x80 );  sclk\_set( 1 );  d = d << 1;  sclk\_set( 0 );  }  } | void lcd5510\_command(  unsigned char d  ){  dc\_set( 0 );  sce\_set( 0 );  lcd5510\_send\_byte( d );  sce\_set( 1 );  }  void lcd5510\_data(  unsigned char d  ){  dc\_set( 1 );  sce\_set( 0 );  lcd5510\_send\_byte( d );  sce\_set( 1 );  } |

The pixels on the LCD screen are stored in an array of bytes. Each byte represents 8 vertical pixels . The next byte represent the 8 pixels immediately to the right, etc.

|  |
| --- |
|  |
|  |

The interface to the LCD as we use it does not provide for a way to read the pixel data, hence to set a single pixel we must know the value of the other 7 pixels in its byte. On the LPC1114 we keep a copy of the pixel data. For 48 \* 84 pixels we need 4032 bits, which is 504 bytes. The lcd5510\_write function checks whether the x and y coordinates are valid, calculates the byte within the array and the bit position that must be changed, changes it, and then writes that byte out to the LCD.

|  |
| --- |
| Unsigned char lcd5510\_buf[ 504 ];  void lcd5510\_write(  unsigned int x,  unsigned int y,  unsigned char d  ){  unsigned int a = x + ( y / 8 ) \* 84;  unsigned int m = 1 << ( y % 8 );    if(( x >= 84 ) || ( y >= 48 )){ return; }    if( d ){  lcd5510\_buf[ a ] |= m;  } else {  lcd5510\_buf[ a ] &= ~m;  }    lcd5510\_pixels( x, y / 8, lcd5510\_buf[ a ] );  } |

The command lcd5510\_pixels is internal to the LCD library. It writes a single byte of pixels to the byte at x, y. Note that X and Y are byte coordinates, not pixel coordinates. The function issues the two command bytes to the PCD8544 that set the X and Y pointers to the correct values, and then writes the single data byte.

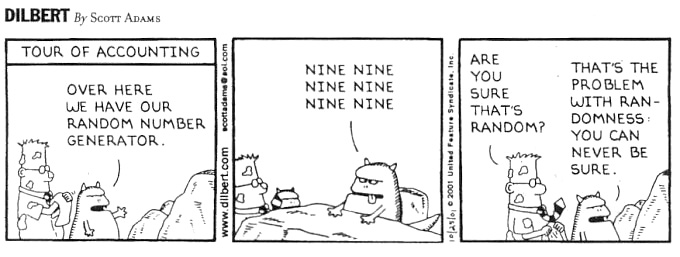
|  |
| --- |
| void lcd5510\_pixels(  unsigned char x,  unsigned char y,  unsigned char d  ){  lcd5510\_command( 0x80 | x );  lcd5510\_command( 0x40 | y );  lcd5510\_data( d );  } |

The lcd\_clear() function could just write all pixels, but that would write each pixel byte 8 times. Instead the function sets the LCD RAM write address to 0, and then write the appropriate number of 0 bytes to both the LCD RAM and the RAM buffer on the microcontroller.

|  |
| --- |
| void lcd5510\_clear( void ){  int i;  lcd5510\_command( 0x80 | 0 );  lcd5510\_command( 0x40 | 0 );  for( i = 0; i < 504; i++ ){  lcd5510\_buf[ i ] = 0;  lcd5510\_data( 0 );  }  } |

The drawing of a line uses a technique that named after the inventor Bressenham. The problem is that a line from for instance 0,0 to 10,7 must, for each step in the x direction, ‘climb’ 0.7 step in the Y direction. But the only option we have is to climb 0 or 1 pixel. To get a good-looking line we must maintain the Y position with more accuracy, and at every X position round the Y to an integer value and draw the resulting pixel. This technique is applicable in other problems too, for instance when we want to make an average-exact tone using a delay routine that is accurate but does not have sufficient resolution.

The demo application draws random lines and periodically clears the display. Nothing inside a computer can be really random, all we can do is cheat by using a deterministic algorithm that generates values that look random.



We use a so-called Linear Congruential algorithm, which produces a sequence of numbers that appear surprisingly random. The values of the two magic numbers are not selected at random, they must satisfy some complex mathematic properties to generate a reasonably random sequence[[8]](#footnote-8). The last returned number is stored in the static variable n. Static in this context means that the variable has a global life-time, but it is visible only to the function in which it is declared.

Note that every time we start the chip and ask for a sequence of numbers we will get the same sequence! We could get a truly random sequence by starting with a random number for n, but nothing inside our chip will give produce a truly random number to start with. Sources for a reasonably random start number must come from outside the microcontroller chip, for instance be the lower bits of the A/D reading of an LDR, or the lower bits of the duration of some user interaction, for instance how long the user pressed a button in microseconds, modulo 1000 microseconds.

|  |
| --- |
| int rand(){  static int n = 0;  n = n \* 214013 + 2531011;  return ( n >> 16) & 0x7fff;  } |

To get random start and end points for drawing a line the function int random\_in\_range() is used, which returns a number in the range min .. (max-1). It takes a random number (which covers the full range of 16-bit values) and uses the % (modulo) operator to get a number in the interval 0 .. (max – min - 1), to which the min value is added to get a number in the requested range.[[9]](#footnote-9)

|  |
| --- |
| unsigned int random\_in\_range( unsigned int min, unsigned int max ){  unsigned int x = rand();  x = x % ( max - min + 1 );  return min + x;  } |

The main() of the application initializes the timer and the LCD. Then it clears the LCD and draws a frame at the edges. Then it draws 30 random lines, and starts all over. A random point in the drawable plane is selected requesting and X and Y coordinate within the range 0..83 resp. 0..47.

|  |  |
| --- | --- |
| int main( void ){  unsigned int x, y, n, x1, y1;  timer\_init();  lcd5510\_init();  for(;;){  lcd5510\_clear();  for( x = 0; x < 84; x++ ){  lcd5510\_write( x, 0, 1 );  lcd5510\_write( x, 47, 1 );  }  for( y = 0; y < 48; y++ ){  lcd5510\_write( 0, y, 1 );  lcd5510\_write( 83, y, 1 );  }  for( n = 0; n < 30; n++ ){  x = random\_in\_range( 0, 84 );  x1 = random\_in\_range( 0, 84 );  y = random\_in\_range( 0, 48 );  y1 = random\_in\_range( 0, 48 );  lcd5510\_line( x, y, x1, y1, 1 );  await( now() + 500 \* 1000 );  }  }  } |  |

## Using an LPC810 chip

The LPC810 is a smaller cousin of LPC1114 on the DB103 board.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Name | LPC1114FN28 | LPC810M021FN8 |
| Pins (easily usable) | 28 (18) | 8 (2-4) |
| Flash | 32 Kb | 4 Kb |
| RAM | 4 Kb | 1 Kb |
| Max frequency | 48 MHz | 30 MHz |
| Peripherals | A/D, PWM, timers, UARTs | Timers, UART |

The LPC810 can be programmed the same way as the LPC1114, but because it is a bare chip we must make the connections between the DB100 downloader and the LPC810 using a breadboard and wires. Six of the eight pins of the DB100 must be connected. The diagrams show which, and the recommended wire colors to use. Note that (as the colors show) the downloaders TxD pin must be connected to the chips RxD pin and vice versa. Two GPIO pins are left unconnected by the loader and free for you to use: PIO\_0\_2 and PIO\_0\_3. The Bootmode and TxD pins can also be used, but the Bootmode pin must not be pulled low when the chip starts. It could be used to drive a LED that is connected to +3V3, or a switch that is connected to ground.

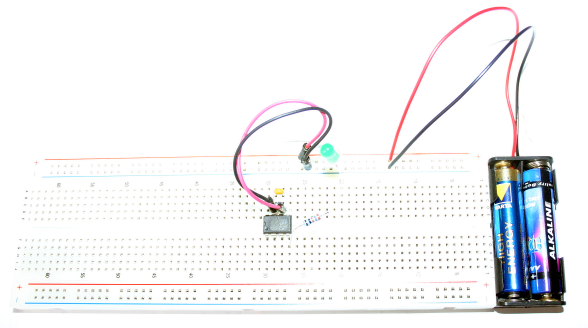
|  |  |
| --- | --- |
|  |  |
| |  |  |  |  | | --- | --- | --- | --- | | Bootmode | Reset | RxD | TxD | | +3V3 |  |  | Ground | | |  |  |  |  | | --- | --- | --- | --- | | RxD GPIO\_0\_0 | Ground | +3V3 | Bootmode GPIO\_0\_1 | | LPC810  O | | | | | Reset | TxD  GPIO\_0\_4 | PIO\_0\_3 | PIO\_0\_2 | |
|  | |

In addition to the connections with the DB100 downloader one 100nF capacitor must be used, directly at the Ground and +3V3 pins of the chip.

The examples directory has an LPC810 project that blinks a LED on PIO\_0\_2. It provides a library with function for using the pins and a simple wait. These functions have C headers, but are implemented in C++.

|  |  |
| --- | --- |
| **Function** | **Use** |
| void pin\_configure\_as\_input( int N ); | Configure pin N as input. |
| void pin\_configure\_as\_output( int N ); | Configure pin N as output. |
| void pin\_set( int N, unsigned char X ); | Set pin N to the boolean value X. The pins must first be configured as output. |
| unsigned char pin\_get( int N ); | Get the level of the pin as Boolean value. The pin must first be configured as input. |
| void wait\_us( int N ); | Wait N microseconds. |

Once you have the LPC810 chip programmed you can remove the DB100 and power it from two 1.5V batteries. Don’t connect the batteries while you have the DB100 loader connected: this would cause the batteries to be charged, which is not something a normal (non-rechargeable) battery is designed for. Also don’t short the battery connections, that can cause the batteries to get very hot.



# Appendix A : hardware set content

|  |  |  |
| --- | --- | --- |
| **N** | **Item** | **Notes** |
| 1 | Plasic box |  |
| 1 | DB100 assembled |  |
| 1 | DB103 assembled, minimum configuration |  |
| 1 | USB A-B cable |  |
| 2 | Solderless breadboard |  |
| 1 | Jumper wires set |  |
| 1 | Potentiometer 1kΩ |  |
| 1 | LCD Nokia5110 |  |
| 1 | Speaker with pins |  |
| 1 | 3-wire analog hobby servo | Brown = ground; Red = power; Yellow = pulse |
| 1 | Keypad 3x4 or 4x4 |  |
| 1 | Battery holder 2 x AAA |  |
| 2 | Batteries AAA |  |
| 1 | Set of colored pieces of paper |  |
| 2 | Pieces of black tube |  |
| On black foam: | | |
| 1 | Chip MCP23017 |  |
| 2 | Chip 74HC595 |  |
| 1 | Chip LPC810 |  |
| 1 | 4-digit LED display DISP-06 |  |
| In plastic bag: | | |
| 10 | Resistor 100 Ω | Color code: brown-black-brown |
| 15 | Resistor 1 kΩ | Color code: brown-black-red |
| 10 | Resistor 10 kΩ | Color code: brown-black-orange |
| 1 | LDR |  |
| 2 | Capacitor 100 nF | Coded: 104 |
| 1 | Capacitor 470 µF | White martking == ground |
| 10 | LED 5mm RED | Flat side / short wire == ground |
| 10 | LED 5mm GREEN | Flat side / short wire == ground |
| 10 | LED 5mm YELLOW | Flat side / short wire == ground |
| 1 | LED 5mm BLUE | Flat side / short wire == ground |
| 1 | LED 5mm IR TSUS5202 | Color is deep blue/purple/black |
| 1 | IR receiver chip TSOP34836 |  |
| 1 | Chip LM335 |  |
| 1 | Chip TL431 |  |
| 2 | ID Coin DS1990A |  |
| 5 | Switch ‘SW-03’ |  |

# Appendix B : DB103 full circuit diagram

|  |  |
| --- | --- |
|  |  |

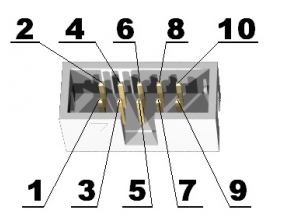
The DB103 board as we use it has the LPC1114 microcontroller IC1, the U$10 connector for the DB100, the two 4k7 pull-up resistors for the I2C pins, the reset switch SW2, and the two 3-pin and 3 8-pin wire cup rows. The solder jumpers J2 and U$16 are placed.

Other components that can be put on the board are:

1. either the lithium coin-battery holder U$9 or the screw-connector CON-1 for an external battery;
2. The slide switch SW1 which selects between the battery and the power from the DB100 (or off, when no DB100 is connected). (The solder bridge J1 must be removed.);
3. One of the FETs T1A or T1B to allow the microcontroller two switch the power to one of the power cups. (The solder bridge U$16 must be removed.);
4. The 2x5 pin UEXT connector can be placed to accept Olimex UEXT compatible peripherals. The resistors R1 and R2 must be placed when the asynchronous serial pins of the UEXT connector must be used. Note that when connected a DB100 will block this use. The resistors will prevent any electrical damage from this conflict. The solder jumpers in the SCL and SDA liens must be placed if the I2C pins of the connector are to be used;
5. An RFM73 radio module can be placed.

# Appendix C : DB103 pin assignment

The table below shows how each GPIO pin of the LPC1114 chip on the DB103 board is connected. The function column states the special function that the pin has in the LPC1114. The RFM73 column refers to the optional RFM73 module that can be fitted next to the reset switch. The MPL10 column refers to the optional 2x5 pin connector that can be fitted next to the programming connector. This connector is compatible with the Olimex UEXT definition.



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PIN** | **function** | **Cup rows** | **RFM73** | **ML10** | **ISP** | **Notes** |
| PIO0\_0 | RESET |  |  |  | X |  |
| PIO0\_1 | BOOT |  |  |  | X | Controls the optional power switch PFET |
| PIO0\_2 | SSEL0 |  | SSEL | 10 |  |  |
| PIO0\_3 |  |  | CE |  |  |  |
| PIO0\_4 | I2C - SCL | A |  | 5 \* |  | Open drain pin; 4k7 pull-up |
| PIO0\_5 | I2C - SDA | A |  | 6 \* |  | Open drain pin; 4k7 pull-up |
| PIO0\_6 |  | A |  |  |  |  |
| PIO0\_7 |  | A |  |  |  |  |
| PIO0\_8 | MISO0 |  | MISO | 7 |  |  |
| PIO0\_9 | MOSI0 |  | MOSI | 8 |  |  |
| PIO0\_10 | SCK0 |  | SCK | 9 |  |  |
| PIO0\_11 | POWER |  |  |  |  |  |
| PIO1\_0 |  | B |  |  |  |  |
| PIO1\_1 |  | B |  |  |  |  |
| PIO1\_2 |  | B |  |  |  |  |
| PIO1\_3 |  | B |  |  |  |  |
| PIO1\_4 |  | C |  |  |  |  |
| PIO1\_5 |  | C |  |  |  |  |
| PIO1\_6 | UART-TXD |  |  | 4 # | X | Downloader RxD |
| PIO1\_7 | UART-RXD |  |  | 3 # | X | Downloader TxD |
| PIO1\_8 |  | C |  |  |  |  |
| PIO1\_9 |  | C |  |  |  |  |

\*: via a solder jumper #: via a 1k resistor

# Appendix D: Nokia 5510 LCD modules connections

Two LCD modules can be used, which are fully software and hardware compatible, but have different pin assignments. The pin names are stenciled on the bottom side of the prints.

|  |  |  |
| --- | --- | --- |
|  | **Pin** | **Function** |
| LED | background |
| SCLK | SPI clock |
| DN (MOSI) | SPI Master Out Slave In |
| D/C | Data / Command |
| RST | Reset |
| SCE | chip select |
| GND | Ground |
| VCC | 3.3V |

|  |  |  |
| --- | --- | --- |
|  | **Pin** | **Function** |
| GND | Ground |
| LED | Background |
| VCC | 3.3V |
| CLK | SPI clock |
| Din | SPI Master Out Slave In |
| D/C | Data / Command |
| CE | chip select |
| RST | reset |

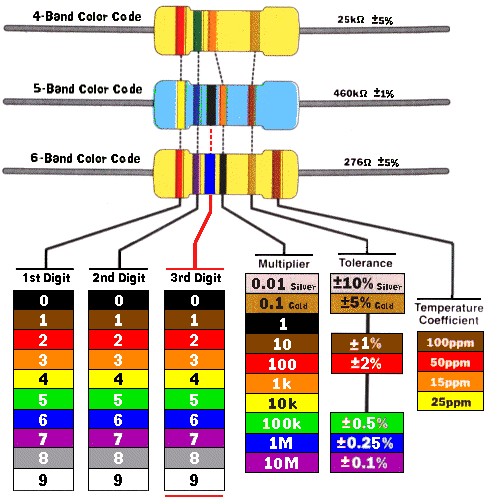
# Appendix E : some (wild?) suggestions for your final project

The final project must be sufficiently challenging to warrant a mark for the course, yet sufficiently easy to be doable. The right balance might vary with the individual capabilities, ambitions and background of each student. If you have something that you have always wanted to automate that might be a good project: your model train, motorbike alarm, garden watering system, room burglary alarm, etc. The use of hardware outside what is provided in the kit is certainly encouraged, but not required. Combining the microcontroller(s) with your PC might also be an idea: gather some data, present it on the PC.

Some (wild?) ideas for your inspiration:

* Have remote fun with the TV (go TV-be-gone war-driving, or annoy your kid brother by switching to the sports channel whenever nickelodeon is selected)
* Automate something you already have (model train?)
* Get some toy, rip out the electronics and replace it with the LPC1114 (check Action & Blokker stores, bring new life to that Furby)
* Some (retro-) game (hit the wumpus, connect-four, breakout, etc); make it multi-palyer by connecting microcontrollers
* Use RF communication
* Calculator (booooring…)
* Metronome (check ‘Take Five or ’‘Golden Brown’: I want 3/4 + the occasional 4/4!)
* James Bond’s worst nightmare (bomb? Laser cutter?)
* “The most useless machine ever” (youtube) – but give its some extra twist
* Colored ping-pong ball sorter (like in Nemo)
* Combine with SW on a PC (logic analyser? Oscilloscope? Weather forecast?)
* DIY barcode scanner
* Laser Tag game

# Appendix F : the resistor color code



The value (resistance) of a resistor is encoded by a number of colored bands. The tolerances are mostly 10% (silver) or 5% (gold). These colors are not used for the digits, so turn the resistor around to keep these colors at the right side. Watch out for the difference between orange and red!

1. The notation A->B means that A is a pointer which points to a structure that has a member B, and the total expression is that member. It can be used both on the left and on the right of an assignment. Don’t worry If you don’t understand that fully, for this course you can use the A->B notations declared by the header as if they were variable. [↑](#footnote-ref-1)
2. More accurately: there must be a .h file and a file that can be compiled to a .o file. This can be a .c file, a.cpp (C++) file, or a .asm (assembler) file. [↑](#footnote-ref-2)
3. Using an absolute path can give problems with some make.exe tools. [↑](#footnote-ref-3)
4. If the capacitor is not included the current drawn by the servo can cause the power supply voltage at the LPC1114 to drop so low that the chip will reset itself. In some circumstances such problems might still occur even with the elco; in that case you could try to power the servo from the two 1.5V batteries. [↑](#footnote-ref-4)
5. When the frequency deviates by 10% the sensitivity of the receiver drops to 50%, which translates to a maximum distance that is reduced to 1/√2 (= 71%) according to the inverse square law. [↑](#footnote-ref-5)
6. You might wonder how can we reset the chip after we have reconfigured the reset pin the to be a normal GPIO? There is only one method left: remove the power. For this reason the DB100 board contains a FET that can remove the power to the DB103 board (with the LPC1114) under control of the PC. Without this feature there would be no way for the serial downloader to reprogram an LPC1114 that has ‘disconnected’ its reset pin. [↑](#footnote-ref-6)
7. The I2C standard allows a slave to do clock stretching, by keeping the clock low until it is ready to deliver the bit. The chip that we use doesn’t use this feature. [↑](#footnote-ref-7)
8. If you realy want to know: the two magic numbers must be relatively prime (share no factor), and all divisors of the mask (sequence length - 1) must be divisors of the magic numbers. Or so my more mathematically inclined colleague said. [↑](#footnote-ref-8)
9. Note that this modulo trick gives a number in the requested range, but this number is not uniformly distributed over the rage, because when the original range of random numbers is folded over the requested range, the last part folded will not cover the entire requested range, making the lower numbers slightly more likely (google: pigeonhole principle). For our application this is no problem, but it can have serious consequences in other applications, for instance in combination with cryptography. [↑](#footnote-ref-9)