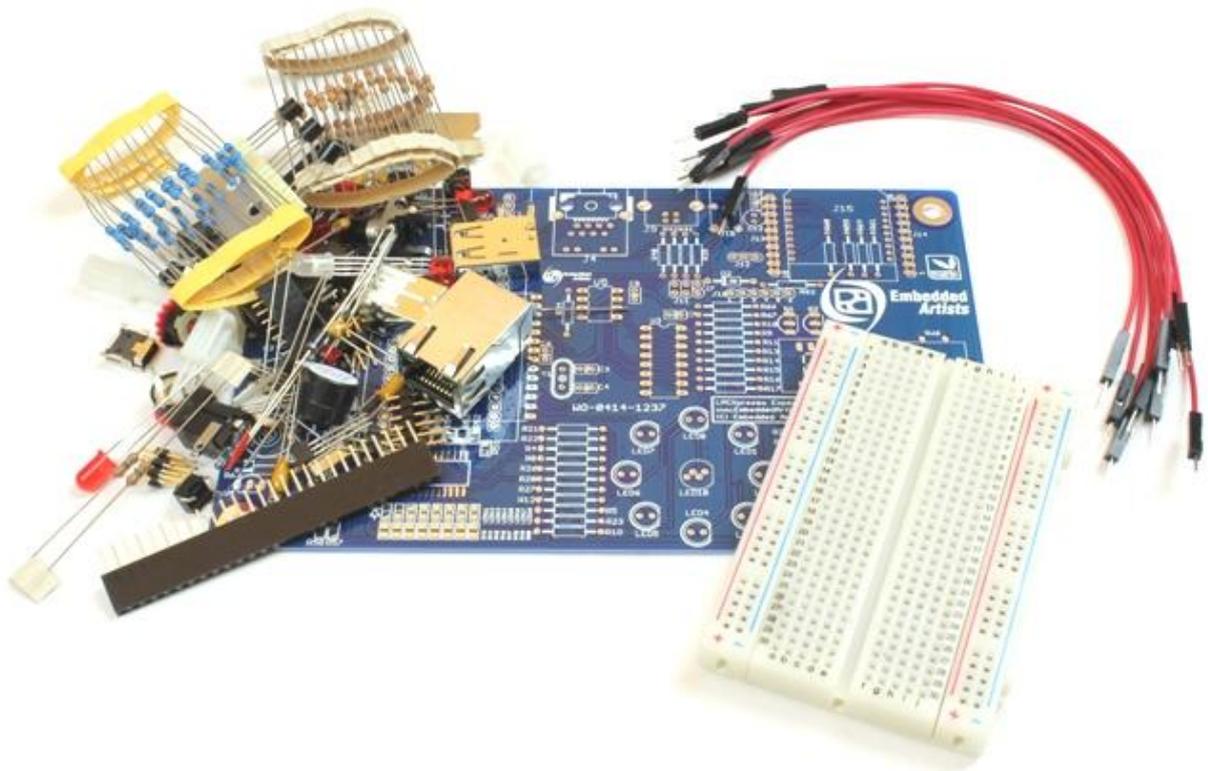


LPCXpresso Experiment Kit User's Guide



Learn embedded programming with NXP's LPC1000 family of Cortex-M0/M3 microcontrollers!

Embedded Artists AB

Davidshallsgatan 16
211 45 Malmö
Sweden

info@EmbeddedArtists.com
<http://www.EmbeddedArtists.com>

Copyright 2013 © Embedded Artists AB. All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Embedded Artists AB.

Disclaimer

Embedded Artists AB makes no representation or warranties with respect to the contents hereof and specifically disclaim any implied warranties or merchantability or fitness for any particular purpose. Information in this publication is subject to change without notice and does not represent a commitment on the part of Embedded Artists AB.

Feedback

We appreciate any feedback you may have for improvements on this document. Please send your comments to support@EmbeddedArtists.com.

Trademarks

All brand and product names mentioned herein are trademarks, services marks, registered trademarks, or registered service marks of their respective owners and should be treated as such.

Table of Contents

1	Document Revision History	7
2	Introduction	8
2.1	Features	8
2.2	ESD Precaution	9
2.3	General Handling Care	9
2.4	Code Read Protection	9
2.5	CE Assessment	9
2.6	Other Products from Embedded Artists	9
2.6.1	Design and Production Services	9
2.6.2	OEM / Evaluation / QuickStart Boards and Kits	10
3	LPCXpresso Experiment Kit	11
3.1	Embedded Systems Programming	12
4	Kit Content	13
5	Powering Options	25
6	Soldering	27
6.1	Component Placement	27
7	Experiments	29
7.1	Preparation	29
7.2	Control a LED	29
7.2.1	Lab 1a: Control LED	30
7.2.2	Lab 1b: GPIO and Bit Masking	35
7.2.3	Lab 1c: Delay Function – LED Flashing	36
7.2.4	Lab 1d: Morse Code	37
7.3	Read a Digital Input	38
7.3.1	Lab 2a: Read Push-button	38
7.3.2	Lab 2b: GPIO and Bit Masking	41
7.3.3	Lab 2c: Logic between inputs and output	41
7.3.4	Lab 2d: Toggling LED	44
7.3.5	Lab 2e: Sampling of Inputs	44
7.4	Control Multiple LEDs	46
7.4.1	Lab 3a: LEDs in Running-One Pattern	46
7.4.2	Lab 3b: Control of Running-One Pattern	47
7.4.3	Lab 3c: Rotary Switch Control of Running-One Pattern	48
7.5	Print Messages	49
7.5.1	Lab 4a: Semihosting and printf()	49
7.5.2	Lab 4b: Semihosting Performance Test	51
7.5.3	Lab 4c: Printing Events	51
7.5.4	Lab 4d: Reading from the Console	51

7.6	Read an Analog Input	53
7.6.1	Lab 5a: Read Trimming Potentiometer	53
7.6.2	Lab 5b: Event Threshold	56
7.6.3	Lab 5c: Read Light Sensor	56
7.6.4	Lab 5d: ADC Noise Test	57
7.7	Pulse Width Modulation	58
7.7.1	Lab 6a: PWM Control of a LED	58
7.7.2	Lab 6b: PWM Control of a LED, cont. 1	59
7.7.3	Lab 6c: PWM Control of a LED, cont. 2	59
7.7.4	Lab 6d: PWM Control of two LEDs	60
7.8	Control an RGB-LED	61
7.8.1	Lab 7a: Test RGB-LED	61
7.8.2	Lab 7b: Control RGB-LED	62
7.9	Control a 7-segment Display	63
7.9.1	Lab 8a: Test 7-segment Display	64
7.9.2	Lab 8b: Control 7-segment Display	64
7.9.3	Lab 8c: Control 7-segment Display, cont.	66
7.9.4	Lab 8d: Control Dual Digit 7-segment Display	66
7.9.5	Lab 8e: Control 7-segment Display via Shift Register	68
7.10	Work with a Timer	71
7.10.1	Lab 9a: Create Exact Delay Function	71
7.11	PWM via a Timer	72
7.11.1	Lab 10a: Control RGB-LED	74
7.11.2	Lab 10b: Buzzer and Melodies	74
7.11.3	Lab 10c: Control a Servo Motor	75
7.12	Work with a Serial Bus – SPI	78
7.12.1	Lab 11a: Access Shift Register	81
7.12.2	Lab 11b: Control 7-segment Display	82
7.12.3	Lab 11c: Access SPI E2PROM	82
7.13	Work with Interrupts	87
7.13.1	Lab 12a: Generate IRQ via GPIO	89
7.13.2	Lab 12b: Timer IRQ	90
7.13.3	Lab 12c: Timer IRQ with Callback	91
7.13.4	Lab 12d: Nested Interrupts	92
7.13.5	Lab 12e: Control Dual Digit 7-segment Display	93
7.14	Work with a Serial Bus – I2C	94
7.14.1	Lab 13a: Solder Surface Mounted Components	95
7.14.2	Lab 13b: Read LM75 Temperature Sensor	96
7.14.3	Lab 13c: Control LEDs via PCA9532	97
7.15	Work with a Serial Bus – UART	100
7.15.1	Lab 14a: Transmitting and Receiving via the UART	106
7.15.2	Lab 14b: Direct printf() to UART	106
7.15.3	Lab 14c: Interrupt driven UART handling and ring buffers	107
7.16	Extra: Work with RF-module	112
7.16.1	Lab 15a: XBee™ RF-Module	113
7.16.2	Lab 15b: GPS Receiver	117

7.17	Extra: Work with Serial Expansion Connector	123
7.17.1	Lab 16a: 128x128 OLED Graphical Display	123
7.18	Extra: Work with USB Device	126
7.18.1	Lab 17a: USB Device – HID	126
7.18.2	Lab 17b: USB Device – Mouse HID	127
7.19	Extra: Work with USB Host	128
7.19.1	Lab 18a: USB Host	128
7.20	Extra: Work with Ethernet Interface	129
7.20.1	Lab 19a: easyWeb Web Server	129
7.20.2	Lab 19b: lwIP TCP/IP Stack, Web Server and FreeRTOS	130
7.21	Differences between LPCXpresso LPC111x and LPC1114 in DIL28	133
8	Projects	135
8.1	Interface a Color Sensor	135
8.2	Interface a Real-time Clock (RTC)	135
8.3	Interface a GPS Module	135
8.4	Interface an SD/MMC Memory Card	135
8.5	Interface an Accelerometer and Gyro	135
8.6	Control a LED Matrix	136
8.7	Create a Game with Display + Accelerometer or Gyro	136
8.8	Create General Menu System for a Display	136
8.9	Retrieve Information from Web Servers	136
8.10	USB Mouse Emulation	136
8.11	Registry in E2PROM	137
8.12	Real-Time Dynamic Data with JAVA Applet	137
8.13	Multiplayer Game via RF-module	137
8.14	Home Alarm System	137
8.15	Polyphonic Audio Generation	138
8.16	Audio Processing	138
8.17	Home Automation	138
8.18	Control a Robot	138
8.19	RS-485 Network	138
8.20	Interface an FPGA/CPLD Chip	138
8.21	Analog Electronic Experiments	138
9	LPCXpresso IDE – How to get Started	139
9.1	Importing Projects	139
9.2	Working with a Project and Compiling	141
9.3	Debugging a Project and Downloading	142
9.3.1	Downloading Just Code	146
9.4	Create own Projects by Copy Existing Project	150
9.5	Common Problems	151
9.5.1	Error message: Failed on chip setup	152

10 Further Information

153

1 Document Revision History

<i>Revision</i>	<i>Date</i>	<i>Description</i>
PA1	2012-07-16	Work in progress.
PA2	2013-01-14	Work in progress.
PA3	2013-01-25	First version to be released. All experiments are still not complete.
PA4	2013-01-29	Minor corrections/clarifications.
PA5	2013-02-25	Completed section 7.9 - 7.10.
PA6	2013-03-19	Completed section 7.11-7.14. Cleanup in variable declarations in code fragments. Added instructions about creating driver structured source code.
PA7	2013-04-08	Completed section 7.15. Changed all code fragments to use predefined typedefs for variable declaration. Minor corrections.
PA8	2013-06-13	Completed section 7.16-7.20. Minor corrections/clarifications.

2 Introduction

Thank you for buying Embedded Artists' *LPCXpresso Experiment Kit* designed to work with NXP's ARM Cortex-M0/M3 LPCXpresso target boards.

This document is a User's Guide that describes the *LPCXpresso Experiment Kit* that describes hardware as well as software related to the kit.

2.1 Features

The kit has been created as a guided tour to learn embedded programming with NXP's LPC1000 microcontroller family with Cortex-M0/M3 cores from ARM. The experiments can be performed on a breadboard for maximum flexibility and ease of use. It is also possible to solder the components to a printed circuit board (pcb) and learn soldering at the same time.

Components included in the kit are:

- 8x LEDs
- 2x Trimming potentiometers
- 7x push-buttons
- RGB-LED
- Light sensor (analog)
- Temperature sensor (analog)
- 7-segment LED, dual digit
- E2PROM with SPI interface
- Temperature sensor with I2C interface (only for pcb mounting)
- Piezo buzzer
- Rotary quadrature encoder (only for pcb mounting)
- Shift register
- I2C ports expander (PCA9532, only for pcb mounting)
- USB Host connector (only for pcb mounting)
- USB Device connector (only for pcb mounting)
- RJ45 connector for Ethernet (only for pcb mounting)
- 14-pos serial expansion connector, for interface to for example graphical displays
- 3x servo connectors. Note that servos are not included.
- XBee™ compatible socket (for ZigBee and WiFi modules). Note that RF module is not included.
- LPC1114 in DIL28 package, with 12MHz crystal and SWD connector (only for pcb mounting)
- Local +3.3V voltage regulator
- Miscellaneous resistors, capacitors, transistors and connectors
- Breadboard with cables
- Naked PCB

2.2 ESD Precaution

Please note that the *LPCXpresso Experiment Kit* come without any case/box and all components are exposed for finger touches – and therefore extra attention must be paid to ESD (electrostatic discharge) precaution.

Always work with the LPCXpresso Experiment Kit in a place with proper ESD protection.

Avoiding electrostatic discharge is all about having the same electric potential and to avoid building up charges between different areas where you work. This is easily accomplished by having a conductive surface on your workbench and connecting yourself with this surface via a wrist wrap.



Note that Embedded Artists does not replace boards that have been damaged by ESD.

2.3 General Handling Care

Handle the *LPCXpresso Experiment Kit* and all included components with care. The board is not mounted in a protective case/box and is not designed for rough physical handling. Connectors and components can wear out after excessive use. The *LPCXpresso Experiment Kit* is designed for prototyping use, and not for integration into an end-product.

2.4 Code Read Protection

The LPC1000 family has a Code Read Protection function (specifically CRP3, see datasheet for details) that, if enabled, will make the microcontroller impossible to reprogram (unless the user program has implemented such functionality).

Note that Embedded Artists does not replace LPC1000 family chip where the chip has CRP3 enabled. It's the user's responsibility to not invoke this mode by accident.

2.5 CE Assessment

The *LPCXpresso Experiment Kit* is CE marked. See separate *CE Declaration of Conformity* document.

The *LPCXpresso Experiment Kit* is a class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

EMC emission test has been performed on the *LPCXpresso Experiment Kit*. Standard interfaces like Ethernet, USB, serial have been in use. Connecting other devices to the product via the general expansion connectors may alter EMC emission. It is the user's responsibility to make sure EMC emission limits are not exceeded when connecting other devices to the general expansion connectors of the *LPCXpresso Experiment Kit*.

Due to the nature of the *LPCXpresso Experiment Kit* – an evaluation board not for integration into an end-product – fast transient immunity tests and conducted radio-frequency immunity tests have not been executed. Externally connected cables are assumed to be less than 3 meters. The general expansion connectors where internal signals are made available do not have any other ESD protection than from the chip themselves. Observe ESD precaution.

2.6 Other Products from Embedded Artists

Embedded Artists have a broad range of LPC1000/2000/3000/4000 based boards that are very low cost and developed for prototyping / development as well as for OEM applications. Modifications for OEM applications can be done easily, even for modest production volumes. Contact Embedded Artists for further information about design and production services.

2.6.1 Design and Production Services

Embedded Artists provide design services for custom designs, either completely new or modification to existing boards. Specific peripherals and I/O can be added easily to different designs, for example,

communication interfaces, specific analog or digital I/O, and power supplies. Embedded Artists has a broad, and long, experience in designing industrial electronics in general and with NXP's LPC1000/2000/3000/4000 microcontroller families in specific. Our competence also includes wireless and wired communication for embedded systems. For example IEEE802.11b/g (WLAN), Bluetooth™, ZigBee™, ISM RF, Ethernet, CAN, RS485, and Fieldbuses.

2.6.2 OEM / Evaluation / QuickStart Boards and Kits

Visit Embedded Artists' home page, www.EmbeddedArtists.com, for information about other *OEM / Evaluation / QuickStart* boards / kits or contact your local distributor.

3 LPCXpresso Experiment Kit

The *LPCXpresso Experiment Kit* has been created as a guided tour to learn embedded programming with NXP's LPC1000 microcontroller family with Cortex-M0/M3 cores from ARM. The experiments can be performed on a breadboard for maximum flexibility and ease of use. It is also possible to solder the components to a printed circuit board (pcb) and learn soldering at the same time. Figure 1 illustrates the two ways of working with the kit. To the left, all components have been soldered to the pcb and the LPCXpresso board is mounted in a socket on the pcb. To the right, a bread board is used and wires connect directly between the bread board and the LPCXpresso board. Note that the LPCXpresso board is not included in the normal *LPCXpresso Experiment Kit*.

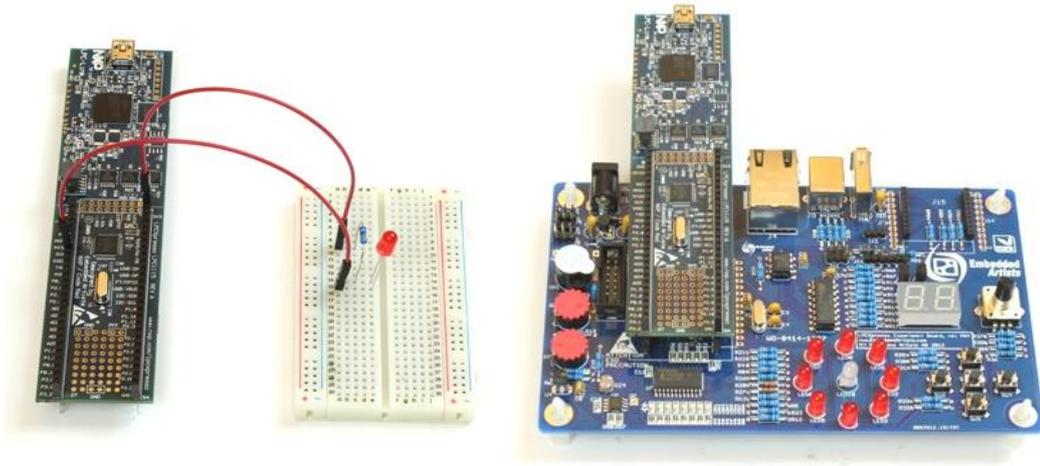


Figure 1 – Breadboard Experiments and Working with PCB

The kit is based on the LPC1000 LPCXpresso evaluation boards, which is a whole family of boards. **All experiments are based around the LPCXpresso LPC1115/1114 board** unless otherwise noted. The term *LPC111x* will be used for the rest of the document to indicate both LPC1115 and LPC1114. Some of the experiments (Ethernet and USB related) are based on the LPCXpresso LPC1769 board. It is also possible to work with the LPC1114 in DIL28 package, which is a breadboard friendly package.

The suggested work flow is as follows: first start with performing the experiments with a group of components on the bread board together with an LPCXpresso board. When done with the experiments, solder the components to the pcb. Continue with the next group of components. Some components only work on the pcb, simply because they do not fit into the bread board. Perform the experiments related to these components when they have been soldered to the pcb. There are of course other ways of working, for example soldering all components to the pcb at the end of all experiments or work separately with the LPC1114 in DIL28 package instead of an LPCXpresso board. Note that in the latter case, an LPC-Link™ is needed to program the LPC1114. The LPC-Link is the “debugger half” of an LPCXpresso board.

The LPC111x is built around a Cortex-M0™ core from ARM and the LPC1769 has a Cortex-M3™ core. Most things addressed with the experiments are general to all microcontrollers and embedded systems programming in general. The details are however slightly different between different microcontrollers, for example the different functionality and registers in the on-chip peripherals.

After having worked with the *LPCXpresso Experiment Kit*, and completed the experiments, you will have gained several competences at basic level:

- embedded programming
- professional debugging techniques
- microcontrollers and how they interact with their environment

- electronic design in general
- how to work with a breadboard
- how to solder

It is assumed that you know how to program in C. You do not have to be an experienced user but at least know about the basics. If not, the Internet is full of ANSI-C tutorials. A good start can be https://en.wikibooks.org/wiki/C_Programming.

The program development environment (also called Integrated Development Environment – IDE, for short) used is the **LPCXpresso IDE**, which is a Eclipse-based IDE, a GNU C-compiler, linker, libraries and an enhanced GDB debugger. For more information see [5].

3.1 Embedded Systems Programming

Embedded systems programming is truly multi-disciplinary. An engineer must master many knowledge areas in order to do a good job. There are at least five of these areas:

- 1) General programming knowledge
(C, algorithms and data structures, understanding the development environment, debugging techniques, safe programming styles, version handling, documentation, etc.)
- 2) Knowledge about programming close to the hardware / Firmware programming
(interrupts, memory mapped accesses for control registers, types of memories, etc.)
- 3) Knowledge about the specific hardware
(details about microcontroller used incl. all peripherals, I/O, communication interfaces, etc.)
- 4) Application programming
(real-time operating systems, program frameworks, user interfaces, drivers, logging, field updates, boot loader structures, factory calibration/settings, configuration management, communication protocols, graphical programming, security, etc.)
- 5) Last but not least, the domain knowledge – the functional that the product under development shall implement.

When working through the experiments in the LPCXpresso Experiment Kit you will increase your knowledge in the first three areas.

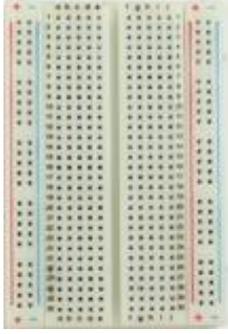
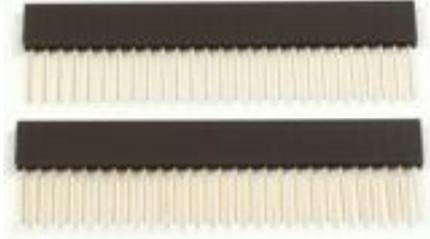
Enjoy working with the LPCXpresso Experiment Kit!

4 Kit Content

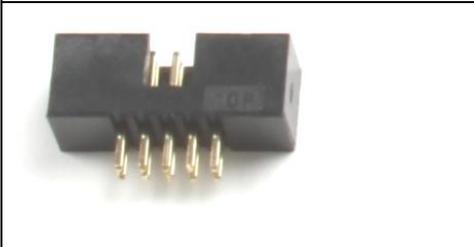
In this chapter we will take a closer look at the different components included in the *LPCXpresso Experiment Kit*.

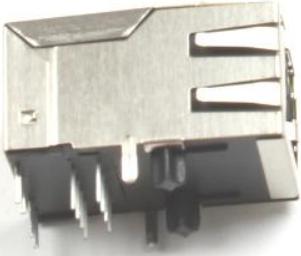
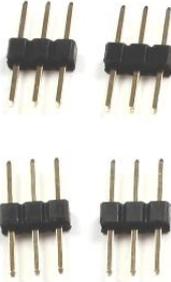
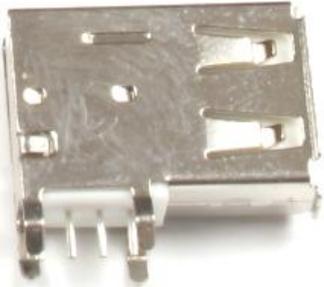
The table below contains photos and a description of all components in order to simplify identification. Note that photos are only typical in the sense that they illustrate the components typical visual appearance. Exact appearance can differ for the components in the kit that you have received. The number of components shown in a picture can also differ from delivered quantity.

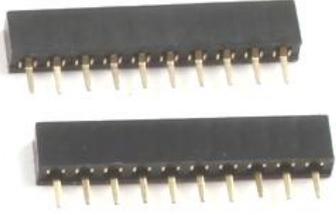
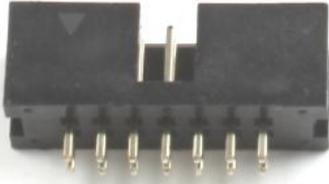
Most components are specified with a Digikey or Mouser equivalent. If a component gets damaged, a new one can typically be ordered from Digikey, Mouser or any preferred component distributor. The Digikey/Mouser number is just to get the key data of the component. The actual components in the component kit might very well be of different brands.

Component	Description	Note
	Breadboard http://en.wikipedia.org/wiki/Breadboard	Digikey: 438-1109-ND Mouser: 854-BB400T
	Cables, male-to-male http://en.wikipedia.org/wiki/Jump_wire	Prototype cables can be ordered from Embedded Artists web shop in 50 pcs packages (EA-ACC-017).
	Connectors for LPCXpresso board 11mm long pins	There is another pair of headers that looks very similar. This pair of connectors has longer pins. The other pair has shorter pins. This pair of connectors shall be soldered to an LPCXpresso board to make it <i>experiment friendly</i> – make it simple to connect cables to the pins. There is no distributor equivalent for this component.

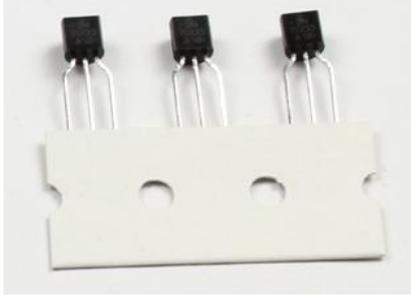
	<p>Tantal capacitor C1, C2, C12 22μF</p> <p>http://en.wikipedia.org/wiki/Tantalum_capacitor</p>	<p>This component is polarized. One of the two pins is longer than the other. This is the positive side. There is also a small plus sign printed on the components on the long pin side.</p> <p>AVX: TAP226K010SCS Digikey: 478-1874-ND Mouser: 581-TAP226K010SCS</p>
	<p>Ceramic capacitor C3, C4 18pF</p> <p>http://en.wikipedia.org/wiki/Ceramic_capacitor</p>	<p>The printed numbers on this component is "180".</p> <p>This is not a polarized component.</p> <p>Murata: RPE5C2A180J2P1Z03B Digikey: 490-3632-ND Mouser: 81-RPE5CA180J2P1Z03B</p>
	<p>Ceramic capacitor C5, C6, C7, C8, C9, C13 100nF</p> <p>http://en.wikipedia.org/wiki/Ceramic_capacitor</p>	<p>The printed numbers on this component is "104".</p> <p>This is not a polarized component.</p> <p>Kemet: C320C104K5R5TA Digikey: 399-4264-ND Mouser: 80-C320C104K5R</p>
	<p>Ceramic capacitor C10, C11 100nF SMT</p> <p>http://en.wikipedia.org/wiki/Ceramic_capacitor</p>	<p>This is a surface mounted component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>This is not a polarized component.</p> <p>Murata: GRM21BR71E104KA01L Digikey: 490-1673-1-ND Mouser: 81-GRM40X104K25L</p>

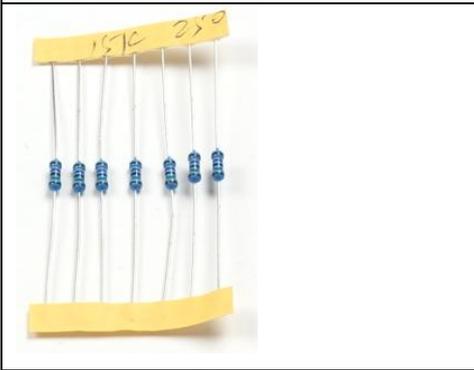
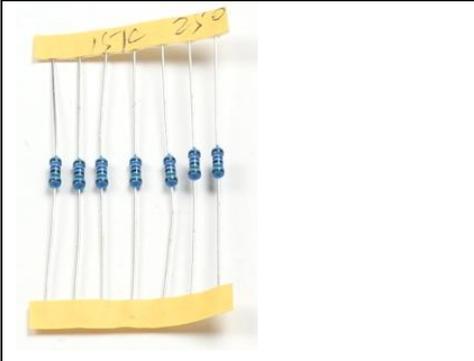
	<p>Schottky diode D1, D2 1N5817</p> <p>http://en.wikipedia.org/wiki/Semiconductor_diode</p> <p>http://en.wikipedia.org/wiki/Schottky_diode</p>	<p>This component is polarized. There is a ring on one pin-side of the components (upper side in the picture). This is the cathode of the diode. The other side (bottom side) is the anode.</p> <p>Diodes Inc: 1N5817-T Digikey: 1N5817DICT-ND Mouser: 621-1N5817</p>
	<p>Stand-offs H1, H2, H3, H4</p>	<p>These stand-offs are mounted in each corner of the pcb.</p> <p>AVC: BS-13S Any standard stand-off for 4mm holes will work.</p>
	<p>Power jack J1</p>	<p>This component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>CUI Inc: PJ-102A Digikey: CP-102A-ND</p>
	<p>Connectors for LPCXpresso board J2</p>	<p>There is another pair of headers that looks very similar. This pair of connectors has shorter pins. The other pair has longer pins.</p> <p>This pair of connectors shall be soldered to the pcb as a socket to the LPCXpresso board.</p> <p>Sullins: PPTC271LFBN-RC Digikey: S7025-ND</p>
	<p>Debug connector J3</p>	<p>This component and can only be soldered to the pcb (i.e., not used on the bread board). Pin 1 is in the top/upper left corner in the picture.</p> <p>There is no distributor</p>

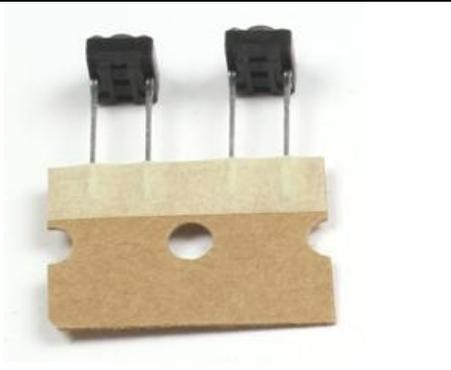
		equivalent for this component.
	RJ45, Ethernet connector J4	This component and can only be soldered to the pcb (i.e., not used on the bread board). Stewart: SI-50170-F Digikey: 380-1103-ND
	Pin list, 1x3 J5, J6, J8, J12	Sullins: PEC03SAAN Digikey: S1012E-03-ND
	Pin list, 2x3 J7 and J11 combined	This component and can only be soldered to the pcb (i.e., not used on the bread board). Sullins: PEC03DAAN Digikey: S2012E-03-ND
	USB-B connector J9	This component and can only be soldered to the pcb (i.e., not used on the bread board). TE Connectivity: 292304-2 Digikey: A98573-ND Mouser: 571-292304-2
	USB-A connector J10	This component and can only be soldered to the pcb (i.e., not used on the bread board). TE Connectivity: 292336-1 Digikey: 292336-1-ND Mouser: 571-292336-1

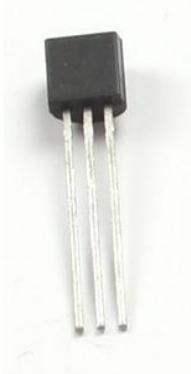
	<p>socket connector for wireless module J15</p>	<p>This component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>Sullins: NPPN101BFCN-RC Digikey: S5751-10-ND</p>
	<p>Shrouded pin list, 2x7 J16</p>	<p>This component and can only be soldered to the pcb (i.e., not used on the bread board). Pin 1 is in the top/upper left corner in the picture.</p> <p>Sullins: SBH11-PBPC-D07-ST-BK Digikey: S9170-ND</p>
	<p>USB mini-B connector J17</p>	<p>This component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>Hirose: UX60-MB-5ST Digikey: H2959CT-ND Mouser: 798-UX60-MB-5ST</p>
	<p>Pin list, 1x6 J18</p>	<p>Sullins: PEC06SAAN Digikey: S1012E-06-ND</p>
	<p>LEDs LED1-LED8</p> <p>http://en.wikipedia.org/wiki/Led</p>	<p>This component is polarized. One of the two pins is longer than the other. This is the positive side, the anode. There is also a small cut on the side of the plastic package. This is on the short pin side, which is the negative side, the cathode.</p> <p>Any 5mm LED with V_f around 1.7V and 150mcd at 20mA current will work, for example: Digikey: 1080-1136-ND</p>

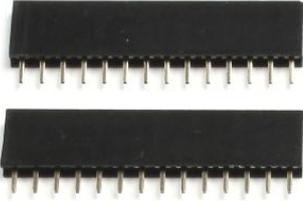
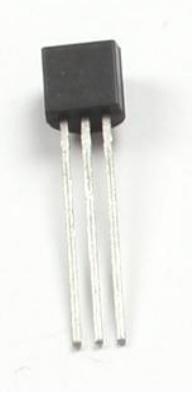
	<p>RGB-LED LED10</p> <p>http://en.wikipedia.org/wiki/Led</p>	<p>This component is polarized. There is a small cut on one side of the plastic package. In the component picture to the left, the cut is on the left side of the package.</p> <p>From left to right the four pins in the picture are:</p> <p>Red-LED cathode All LEDs anode (positive side) Green-LED cathode Blue-LED cathode</p> <p>Harvatek: HT-333RGBW-A Any RGB-LED with common anode and a low value of blue LED V_f (around 3.2V) will work.</p>
	<p>7-segment LED, dual digit LED9</p> <p>http://en.wikipedia.org/wiki/7-segment_display</p>	<p>This component is polarized. Pin 1 is in the lower left corner in the picture to the left.</p> <p>Lite-On Inc: LTD-4608JF Digikey: 160-1536-5-ND Mouser: 859-LTD-4608JF</p>
	<p>LEDs LED11-LED18, SMT</p> <p>http://en.wikipedia.org/wiki/Led</p>	<p>This is a surface mounted component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>This component is polarized. There are green marks on the cathode side.</p> <p>Harvatek: HT17-2102SURC Possible substitute is Kingbright: APT2012SURCK Digikey: 754-1133-1-ND Mouser: 604-APT2012SURCK</p>

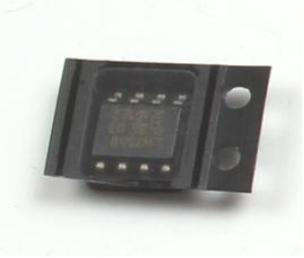
	<p>PNP transistor, BC557B Q1, Q2, Q3</p> <p>http://en.wikipedia.org/wiki/Bjt_transistor</p>	<p>This component is polarized. One side of the plastic package is flat and the other side is rounded. When mounting this component make sure it is turned correctly.</p> <p>ON Semiconductor: BC557BRL1G Digikey: BC557BRL1GOSCT-ND Mouser: 863-BC557BRL1G</p>
	<p>Resistor, 15 Kohm, 7 pcs R1, R3, R35, R36, R41, R42, R59</p> <p>http://en.wikipedia.org/wiki/Resistor</p>	<p>Color: Brown, Green, Black, Red</p> <p>This is not a polarized component.</p> <p>Yageo: MFR-25FBF-52-15K0 Digikey: 15.0KXBK-ND</p>
	<p>Resistor, 0 ohm, 1 pcs R2</p> <p>http://en.wikipedia.org/wiki/Resistor</p>	<p>Color: Black</p> <p>This is not a polarized component.</p> <p>Yageo: ZOR-25-B-52-0R Digikey: 0.0QBK-ND</p>
	<p>Resistor, 330 ohm, 30 pcs R4, R5, R6, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R21, R22, R23, R25, R29, R30, R31, R32, R33, R34, R37, R38, R62, R63, R64</p> <p>http://en.wikipedia.org/wiki/Resistor</p>	<p>Color: Orange, Orange, Black, Black</p> <p>This is not a polarized component.</p> <p>Yageo: CFR-25JB-52-330R Digikey: 330QBK-ND</p>

	<p>Trimming potentiometer, 22 Kohm, 2 pcs R7, R20</p> <p>http://en.wikipedia.org/wiki/Potentiometer</p>	<p>10Kohm equivalent from Bourns Inc.: 3352E-1-103LF Digikey: 3352E-103LF-ND</p>
	<p>Photo resistor, 1 pcs R24</p> <p>http://en.wikipedia.org/wiki/Photo_resistor</p>	<p>This is not a polarized component.</p> <p>Advanced Photonix: PDV- P9002-1 Digikey: PDV-P9002-1-ND</p>
	<p>Resistor, 220 ohm, 2 pcs R27, R28</p> <p>http://en.wikipedia.org/wiki/Resistor</p>	<p>Color: Red, Red, Black, Black</p> <p>This is not a polarized component.</p> <p>Yageo: FMP100JR-52-220R Digikey: 220WCT-ND</p>
	<p>Resistor, 1.5 Kohm, 8 pcs R26, R39, R40, R60, R61, R65, R66, R67</p> <p>http://en.wikipedia.org/wiki/Resistor</p>	<p>Color: Brown, Green, Black, Brown</p> <p>This is not a polarized component.</p> <p>Yageo: FMP100JR-52-1K5 Digikey: 1.5KWCT-ND</p>

	<p>Resistor, 2 Kohm, 16 pcs R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58</p> <p>http://en.wikipedia.org/wiki/Resistor</p>	<p>This is a surface mounted component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>This is not a polarized component.</p> <p>Panasonic: ERJ-6ENF2001V Digikey: P2.00KCCT-ND</p>
	<p>Piezo buzzer, 1 pcs SP1</p> <p>http://en.wikipedia.org/wiki/Buzzer</p>	<p>This component is polarized. One pin is longer than the other. The longer pin is the positive side. The top label also indicates this side with a small plus sign.</p> <p>CUI Inc.: CEP-2242 Digikey: 102-1115-ND</p>
	<p>Pushbuttons, 5 pcs SW1-SW5</p>	<p>This component and can only be soldered to the pcb (i.e., not used on the bread board). The reason for this is that the pins are too short to get reliable connection on the bread board. There are two other special switches in the component kit that are suitable for bread board usage.</p> <p>Omron: B3F-1000 Digikey: SW400-ND Mouser: 653-B3F-1000</p>
	<p>Pushbuttons for breadboard, 2 pcs</p>	<p>These switches are for breadboard usage. Note that the pins must be cut to suitable length before mounted in the breadboard.</p> <p>Panasonic: EVQ-11L05R Digikey: P8079SCT-ND Mouser: 667-EVQ-11L05R</p>

	<p>Rotary encoder, 1 pcs SW6</p>	<p>This component can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>Below is without center switch.</p> <p>Panasonic: EVE-GA1F1724B Digikey: P10859-ND Mouser: 667-EVE-GA1F1724B</p>
	<p>Voltage regulator, MCP1700-330, 1 pcs U1</p> <p>http://en.wikipedia.org/wiki/Low-dropout_regulator</p>	<p>This component is polarized. One side of the plastic package is flat and the other side is rounded. When mounting this component, make sure it is turned correctly.</p> <p>Microchip: MCP1700-3302E/TO Digikey: MCP1700-3302E/TO-ND Mouser: 579-MCP1700-3302E/TO</p>
	<p>Microcontroller, LPC1114FN28, 1 pcs U2</p>	<p>This component is polarized. There is a cut in one end of the plastic package, on the short side. This indicates where pin 1 is located. When mounting this component make sure it is turned correctly.</p> <p>NXP: LPC1114FN28/102 Digikey: LPC1114FN28/102,12-ND Mouser: 771-LPC1114FN28/1021</p>

	<p>Headers for U2</p>	<p>This pair of connector headers can (optionally) be soldered to the pcb as a socket for U2. By adding these connectors/headers it is possible to either mount the LPCXpresso board (in J2 headers) or mount U2 in these headers. If J2 headers are mounted but these headers are not, then it is not possible to mount U2.</p> <p>Sullins: PPTC141LFBN-RC Digikey: S7012-ND</p>
	<p>Shift register, 74HC595, 1 pcs U3</p> <p>http://en.wikipedia.org/wiki/Shift_register</p>	<p>This component is polarized. There is a cut in one end of the plastic package, on the short side. This indicates where pin 1 is located – lower left side in the picture to the left. When mounting this component make sure it is turned correctly.</p> <p>NXP: 74HC595N Digikey: 568-1484-5-ND Mouser: 771-74HC595N</p>
	<p>Temperature sensor, MCP9701, 1 pcs U4</p>	<p>This component is polarized. One side of the plastic package is flat and the other side is rounded. When mounting this component make sure it is turned correctly.</p> <p>Microchip: MCP9701-E/TO Digikey: MCP9701-E/TO-ND Mouser: 579-MCP9701-E/TO</p>
	<p>SPI flash, 25LC080, 1 pcs U5</p> <p>http://en.wikipedia.org/wiki/Flash_memory</p>	<p>This component is polarized. There is a cut in one end of the plastic package, on the short side. This indicates where pin 1 is located. When mounting this component, make sure it is turned correctly.</p> <p>Microchip: 25LC080D-I/P Digikey: 25LC080D-I/P-ND Mouser: 579-25LC080D-I/P</p>

	<p>Temperature sensor, LM75, 1 pcs U6</p>	<p>This is a surface mounted component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>This component is polarized. When rotating the components so that the printed text on the package can be read, pin 1 is in the lower left side on the package. When mounting this component make sure it is turned correctly.</p> <p>NXP: LM75BD Digikey: 568-4688-1-ND Mouser: 771-LM75BD118</p>
	<p>I2C port expander, PCA9532, 1 pcs U7</p>	<p>This is a surface mounted component and can only be soldered to the pcb (i.e., not used on the bread board).</p> <p>This component is polarized. When rotating the components so that the printed text on the package can be read, pin 1 is in the lower left side on the package. When mounting this component make sure it is turned correctly.</p> <p>NXP: PCA9532D Digikey: 568-1039-5-ND Mouser: 771-PCA9532D-T</p>
	<p>12MHz HC49 crystal, 1 pcs Y1</p> <p>http://en.wikipedia.org/wiki/Crystal_oscillator</p>	<p>This is not a polarized component.</p> <p>CTS-Freq. Controls: ATS120B Digikey: CTX904-ND</p>

5 Powering Options

There are a couple of different options how to power the experiments. Below is a short list, summarizing the options:

<i>Controller</i>	<i>R2</i>	<i>Power option #1</i>	<i>Power via external +5V supply (J1 or J17)</i>
LPCXpresso board	Do not mount R2	Power via USB connector on LPCXpresso board	Yes, can also be done
LPC1114 in DIL28	Mount R2	-	Yes
mbed	Do not mount R2	Power via mbed USB connector	Yes, can also be done

If the **servo interface**, **USB Host** interface and/or **RF module** are used the board **MUST** be powered via an external +5V supply. Powering via the USB connectors of the LPCXpresso/mbed module is typically not enough.

Below is a more details description. Read through all different options to determine which powering option fits your needs.

- The simplest and most common way is to let the LPCXpresso board generate the +3.3V supply that is needed. This voltage is available on pin 29 on the LPCXpresso expansion connector (see schematic for details). R2 should not be mounted in this case.
 - The LPCXpresso board can supply up to about 100 mA on the +3.3V supply. Note that by turning on all LEDs and activating all features on the board it is possible to consume more than 100 mA.
 - Note that the voltage is not exactly 3.3V, but a Schottky diode forward voltage drop less, so around 3.15V.
- In case the LPCXpresso board is not powered via its USB connector an external +5V DC supply is needed. Connect the external supply to J1 or J17 (as described below).
 - If the internal +3.3V voltage regulator on the LPCXpresso board is used, R2 shall not be mounted. Else R2 shall be mounted (and U1 is the +3.3V regulator in use).
- If current consumption on the +3.3V supply is higher that the LPCXpresso board can provide an external +5V DC supply is needed. This is typically true when working with wireless/RF modules and/or with the USB Host interface (J10 connector). When working with servo motors an external +5V supply is absolutely needed.
 - An external +5V DC supply can connect to J1, which is a 2.1mm power jack with positive center pin. Note that there is no overvoltage protection in the design. Make sure that the connected power supply does not supply more than +5V DC. The current capability of the external +5V DC supply should be in the region of 1-2 Ampere.
 - Connector J17 (mini-B USB connector on the back side of the pcb) can also be used to supply an external +5V DC supply via the USB Host port on a PC/laptop/USB hub.
- When using the LPC1114 in DIL28 package an external +5V DC supply is needed. Feed the +5V via J1 or J17 (as described above) and mount R2 (in order to let U1 be the +3.3V regulator in use).

- When using an mbed module, this module can generate the needed +3.3V supply (supply comes from its own USB connector). R2 should not be mounted in this case.
 - The mbed module can supply much more current on the +3.3V supply than an LPCXpresso board can.
 - In case the mbed module is not powered via its USB connector, it is possible to power it with an external +5V DC supply via connector J1 or J17 (as described above).

6 Soldering

This chapter describes how to solder the components to the naked pcb. Note that when a component has been soldered it can no longer be used for breadboard experiment.

This chapter will not present a full beginner tutorial on soldering, but rather point out how to get started. There are many good soldering tutorials on the Internet, which can easily be found via a Google search. Sparkfun has a good starting guide: <http://www.sparkfun.com/tutorials/354>. They also have a series of guides for soldering SMD (Surface Mounted Device) components: <http://www.sparkfun.com/tutorials/36>.

The following material is requires before you start soldering:

- Temperature regulated soldering iron (in the 30-80 Watt region)
- Thin (0.5-0.75 mm / 20-30 mil) solder with rosin-core and non-corrosive flux
- Damp sponge or brass sponge
- Wire cutter
- Safety glasses

It is also recommended to have a soldering fume extractor (or work in a well ventilated space and have a fan that simply blows away the soldering fumes). In either case, be aware of the health issues with soldering fumes.

6.1 Component Placement

The picture below illustrates the component placement on the pcb. The picture is also available as a PDF where it is possible to search for the component designators.

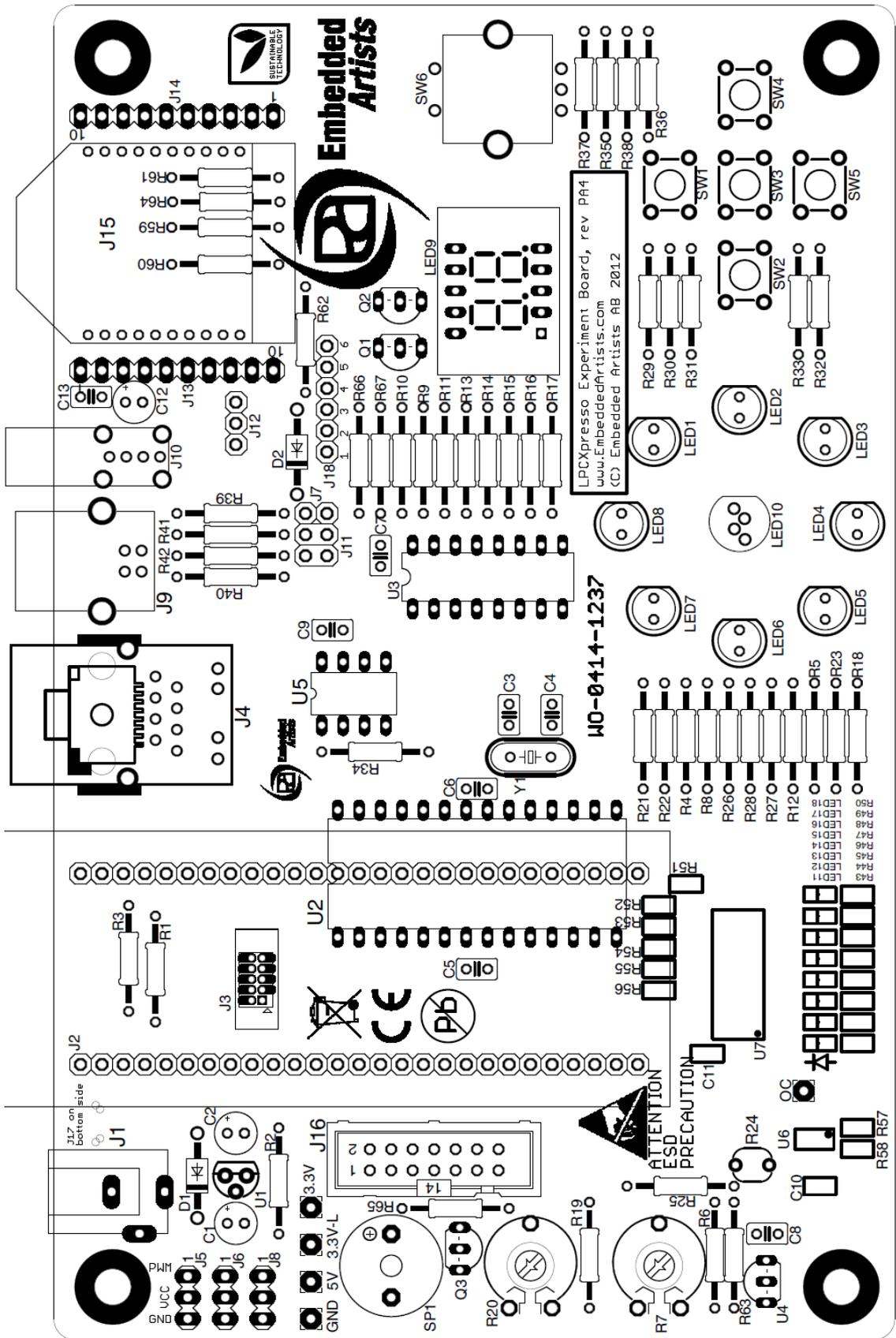


Figure 2 – LPCXpresso Experiment Kit PCB with Component Designators

7 Experiments

This chapter contains the experiments. It is recommended to follow the order of the experiments. It has been compiled to give you the best learning curve. There are multiple small steps in the experiments and they build upon each other. Where appropriate, some theoretical discussions have been added.

All experiments are based around the LPCXpresso LPC111x board unless otherwise noted. Both LPCXpresso LPC1115 and LPC1114 boards are ok to use. Some of the experiments – Ethernet and USB - at the end of the chapter will use the LPCXpresso LPC1769 board. There is also a separate section describing the differences between using the LPCXpresso LPC1115/LPC1114 and the LPC1114 in DIL28 package.

It is recommended to **download the LPC111x User's Manual** from NXP and have it handy. This document is also called UM10398. Many references into this document will be done and this is also part of the learning – how to find the relevant information in a user's manual. It is also recommended to **have the schematic available**.

It is further recommended to **start working with the breadboard**, as opposed to start soldering all components to be pcb. A better time to solder the components is after having completed all the initial, basic experiments.

7.1 Preparation

One preparation is needed before it is possible to start with the experiments. The LPCXpresso LPC111x board must be made *experiment friendly* – a header with female and male connectors shall be soldered to the LPCXpresso board. See picture below for details. Note that there are two sets (of two) of similar 27 position headers in the component kit. It is the headers with long pins that shall be soldered to the LPCXpresso board.



Figure 3 – LPCXpresso Board with Prototype Headers

7.2 Control a LED

In this first experiment you will learn how to control the I/O pins of the LPC111x microcontroller. More specifically you will learn how to control a LED. This first experiment will have a very detailed description since it is the first one and there are a lot of things to learn about how to create, compile, download and debug a program in the LPCXpresso IDE. The level of details in the descriptions will gradually decrease in later experiments.

7.2.1 Lab 1a: Control LED

We will start with controlling LED1 in the schematic, which is found in the schematic on page 4, upper left corner. LED1's cathode is connected to signal GPIO_4-LED-SSEL. LED1's anode is connected to +3.3V via a (current limiting) series resistance. Figure 4 illustrates where LED1 can be found in the schematic. On schematic page 2, we can see that this signal is connected to PIO0_2 on the LPCXpresso LPC1115 board. Figure 5 illustrates where to find the signal and also where to find the +3.3V supply.

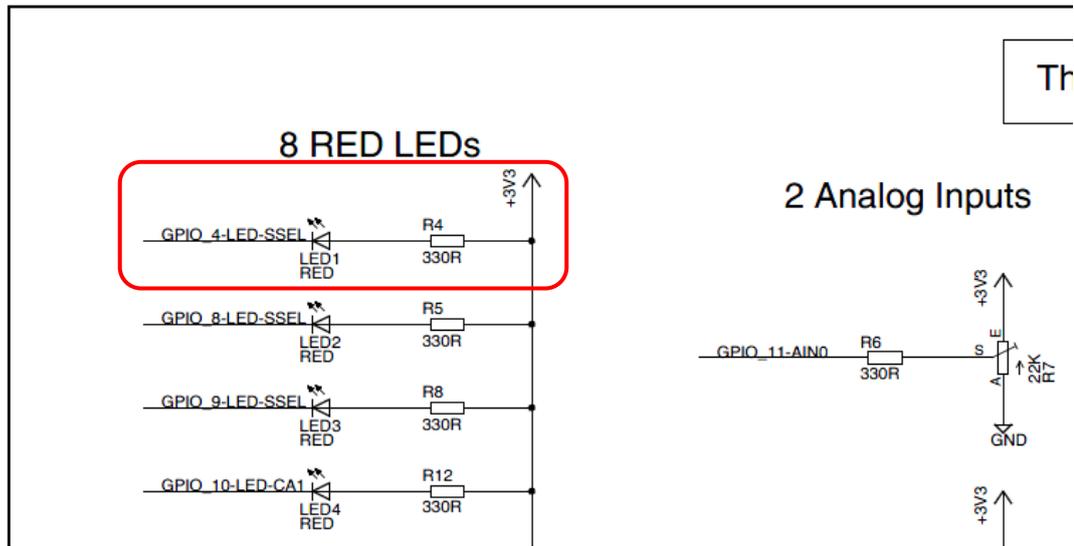


Figure 4 – LED1 on Schematic Page 4

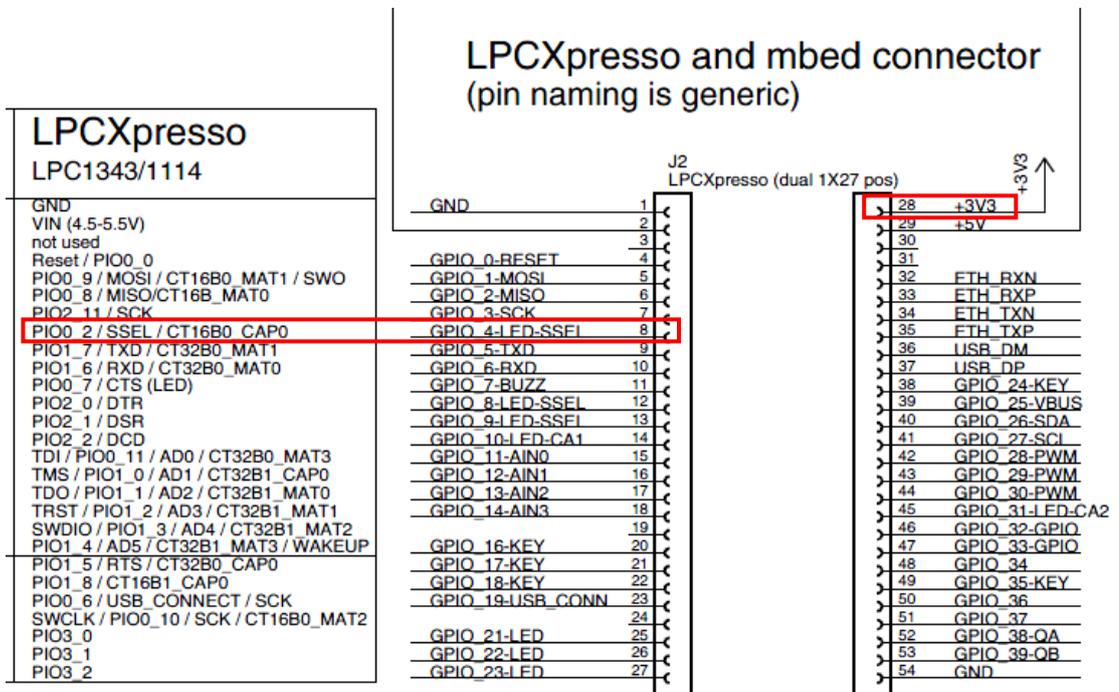


Figure 5 – Signal GPIO_4-LED-SSEL on Schematic Page 2

As a first step, get a LED (representing LED1), a 330 ohm resistor (representing R4), two male-to-male prototype cables and the breadboard from the components bag. Mount these on the breadboard and

connect to the LPCXpresso LPC111x board, as illustrated in Figure 6. Note that only the target processor part of the LPCXpresso board is shown – the black box labeled **LPCXpresso board**. The photo to the left illustrates which part of the real LPCXpresso his black box represents.

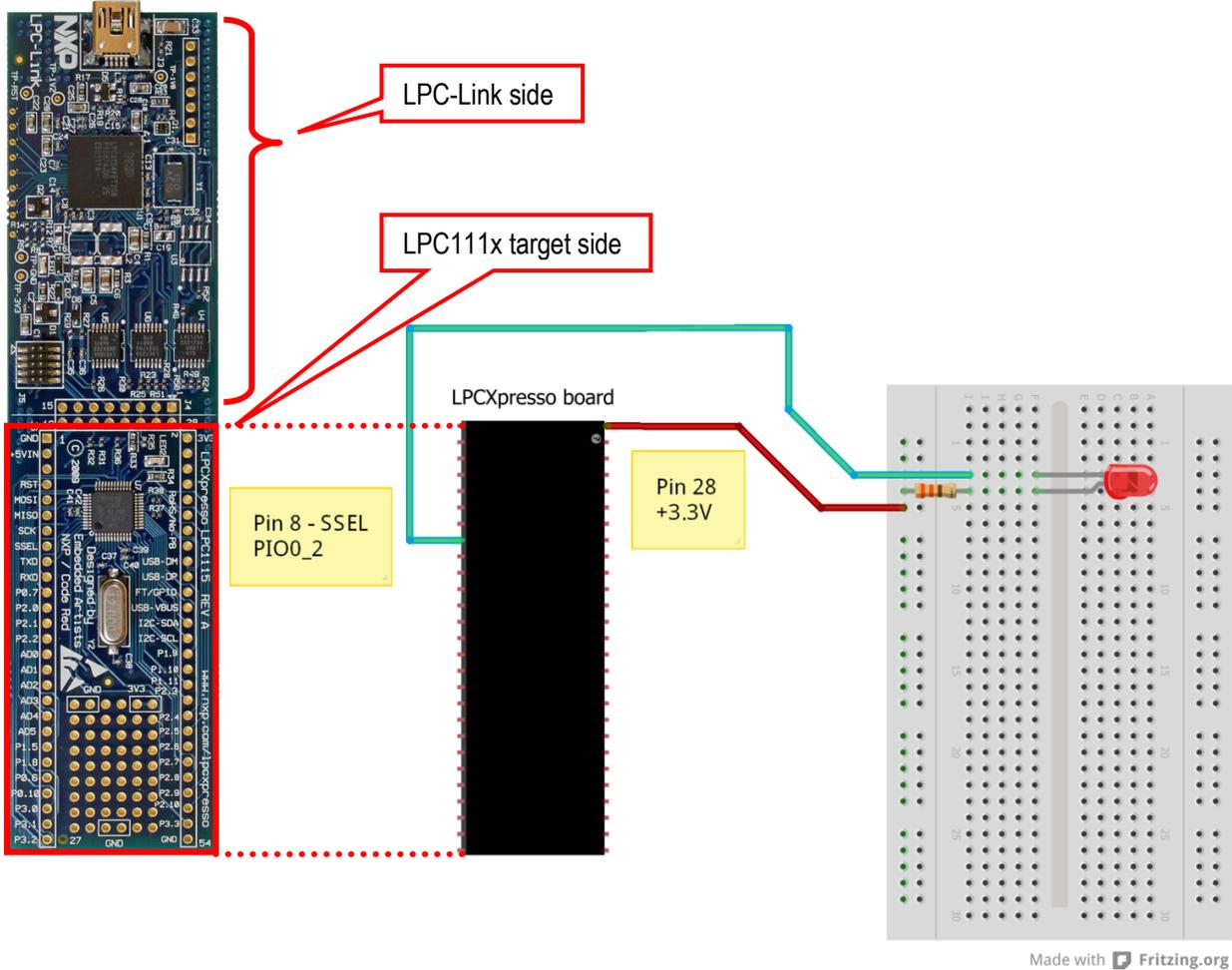


Figure 6 – Breadboard Connections for LED1 (breadboard view)

Figure 7 below illustrates how it can look like in reality. Note that the connections on the breadboard are slightly different than outlined in Figure 6 above. It demonstrates that it is possible to make the connections in many different, yet compatible, ways.

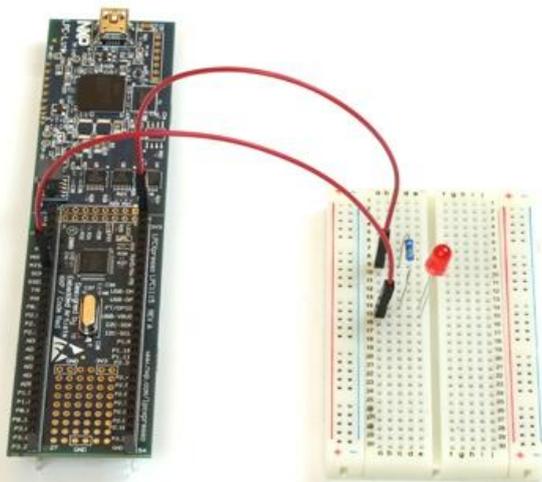


Figure 7 – Breadboard Connections for LED1 (real photo)

The current through the Light Emitting Diode (LED) is limited and controlled by the series resistor. It has to be limited since the voltage drop across the LED is fairly constant. The voltage difference between the LED's forward voltage drop and driving voltage must be absorbed by the series resistor. The current through the LED (and series resistor) can be calculated as $I = (V_{\text{supply}} - V_{\text{leddrop}}) / R$. Different LEDs have different typical current levels. It can be 1, 2, 10, 20 mA for smaller LEDs. Bigger LEDs can have much higher ratings.

The LED forward drop voltage is typically 1.5V for a red LED. Other colors have different forward voltage drops. There are also variations between different brands. Consult the LED's datasheet for details about forward voltage drop and current level. The red LED's included in the component kit has a forward voltage drop of 1.5V and designed for 10mA current. With a 330 ohm series resistor the current is limited to about 5mA, which is OK also. The light intensity at 5mA is acceptable for our (experiment) purposes.

The current level determines the driving method. For moderate levels (typically below 4 mA) most microcontrollers and logic gates can drive the LED directly. This is the method used in our experiments. Some microcontrollers have high-current capacity outputs. The LPC1110 family microcontrollers have a 20 mA output pin (PIO0_7, see datasheet for details).

Almost all output pins have higher current capabilities sinking current than driving current. It is therefore common to connect LEDs like in Figure 4, with the cathode connected to the microcontroller pin. When driving, current is flowing into the micro controller pin (i.e., sinking current).

Another reason for letting the microcontroller drive the LED by sinking current is that most microcontrollers power-up with all pins as inputs with pull-up resistors enabled. This basically means that the pin will be driven high weakly. The LED will not turn on shortly during a power-up. It will be at a known (off) state until the application program controls the LED actively.

If the driving current is higher (> 5 mA) a high-current driver chip can be used, or discrete transistors/mosfets.

A LED is a polarized component, meaning that it matters how the two ends are connected. The two ends are called anode and cathode, respectively. Current flows from anode to cathode, but blocks in the reverse direction. Sometimes the anode is called the positive side and cathode the negative side. The cathode is typically marked somehow on a LED (shorter pin, cut in plastic package, etc).

Mounting a LED the wrong way has no catastrophic result. The result is that the LED will not light (since current through the LED will be blocked). Failing to add the series resistor will have more severe effects, though. Depending on how strong (how much current it can deliver) the power supply is, the current level through the LED can become high enough to destroy the LED. Therefore, be careful to always connect a series resistor with correct resistance value.

The LPC111x is a relatively low pin count processor with only 48 pins. This is true for the package used on the LPCXpresso board. There are other packages with different number of pins for this processor also. The external pins on the chip package are not enough for connecting all internal peripheral units to unique pins. Instead each I/O pin has up to four alternative connections. Read the LPC111x user's manual for more information. You will have to read a lot in this document so you better get started immediately. Have a look in chapter 7 - *LPC1100/LPC1100C/LPC1100L series: I/O configuration* in the LPC111x user's manual for a description of the how the alternative pin functions can be controlled.

Pin PIO0_2 is controlled by register IOCON_PIO0_2. In the description for this register we can see that there are three alternative pin functions:

- PIO0_2, a general purpose input/output, port #0, pin #2

- SSEL0, a control signal for peripheral block SSP
- CT16B0_CAP0, an input signal to 16-bit timer #0

Note that only one functional signal can be connected to the pin at any given point in time. It is however possible to change during program execution. By default, after reset, the register is initialized to PIO0_2, have a pull-up resistor enabled, input hysteresis disabled and to be a standard push/pull GPIO output (if defined as an output). Another register controls the direction of the general purpose digital input/output and this register initialize PIO0_2 to be an input after reset.

Hence, after a reset, PIO0_2 is an input with pull-up resistor enabled. The pin is pulled high weakly but cannot source any larger current. That means that LED1 will be off after reset (because the LED will turn on when PIO0_2 is pulled low and if enough current can sink into the pin).

All LPC111x registers are defined in file: `LPC11xx.h`. It is part of the framework needed to program the LPC111x. Have a look in file `LPC11xx.h`. It is found in the CMSIS library, in the `inc` sub-directory.

What address is register IOCON_PIO0_2 defined as? _____
(you will have to derive the address in several steps – tip: start searching for the LPC_IOCON register at the end of the `LPC11xx.h` file. The register will be accessed as: `LPC_IOCON->PIO0_2`)

Is the derived address the same as in the LPC111x user's manual? _____

Now, have a look in *chapter 12: LPC111x/LPC11Cx General Purpose I/O (GPIO)* in the LPC111x user's manual for a description of how the general purpose I/O functionality is controlled. There is a *GPIO data direction register* that controls the direction of each pin in a port. PIO0_2 belongs to port #0. Bit #2 in register GPIO0DIR controls the direction of the pin. See Figure 8 for details.

12.3.2 GPIO data direction register

Table 174. GPIO0DIR register (GPIO0DIR, address 0x5000 8000 to GPIO3DIR, address 0x5003 8000) bit description

Bit	Symbol	Description	Reset value	Access
11:0	IO	Selects pin x as input or output (x = 0 to 11). 0 = Pin PION_x is configured as input. 1 = Pin PION_x is configured as output.	0x00	R/W
31:12	-	Reserved	-	-

Figure 8 – GPIO Data Direction Register

Register GPIO0DATA holds the current state of the pins in port #0. Bit 2 in this register reflects the state of pin PIO0_2. This is regardless if the pin is an input or output. If a pin is an output the value in GPIOxDATA is driven to the pin.

Table 173. GPIO0DATA register (GPIO0DATA, address 0x5000 0000 to 0x5000 3FFC; GPIO1DATA, address 0x5001 0000 to 0x5001 3FFC; GPIO2DATA, address 0x5002 0000 to 0x5002 3FFC; GPIO3DATA, address 0x5003 0000 to 0x5003 3FFC) bit description

Bit	Symbol	Description	Reset value	Access
11:0	DATA	Logic levels for pins PION_0 to PION_11. HIGH = 1, LOW = 0.	n/a	R/W
31:12	-	Reserved	-	-

Figure 9 – GPIO Data Register

There are also several registers related to interrupt functionality. We will not work with that right now. In later experiments we will return to this.

Note that registers GPIO0DIR and GPIO0DATA are accessed as LPC_GPIO0->DIR and LPC_GPIO0->DATA, respectively.

Below is the two statements needed to first set PIO0_2 to an output and then pull the output low. This will turn the LED on.

```
// Set PIO0_2 as an output
LPC_GPIO0->DIR = LPC_GPIO0->DIR | (0x1<<2);

// Turn LED1 on = set PIO0_2 pin low, i.e., clear bit
LPC_GPIO0->DATA = LPC_GPIO0->DATA & ~(0x1<<2);
```

As seen, each of the registers is first read and then bit #2 is manipulated. In the first statement, bit #2 is set which makes PIO0_2 an output. In the second statement, bit #2 is set to zero. This pulls PIO0_2 low.

Note that all bits in the registers must be read and only the bit of interest shall be manipulated. The shift operation, (0x1<<2), is a good way of writing code. The "<<2" part indicates clearly that it is bit #2 that is manipulated. It is simpler for a reader of the code to quickly see this than to write the constant value 0x04.

Below is an alternative, more compact way of writing the statements. This is a common way to write this kind of statements.

```
// Set PIO0_2 as an output
LPC_GPIO0->DIR |= (0x1<<2);

// Turn LED1 on = set PIO0_2 pin low, i.e., clear bit
LPC_GPIO0->DATA &= ~(0x1<<2);
```

In real, professional programs, it is common to use defines to hide details about hardware manipulation. Below is an example of how this can be done.

```
// Create defines for simpler access of LED1
#define DIR_REG_LED1    LPC_GPIO0->DIR
#define DATA_REG_LED1  LPC_GPIO0->DATA
#define PIO_PIN_LED1    2
#define LED1_ON         DATA_REG_LED1 &= ~(1<<PIO_PIN_LED1)
#define LED1_OFF        DATA_REG_LED1 |= (1<<PIO_PIN_LED1)

// Set PIO0_2 as an output
DIR_REG_LED1 |= (0x1<<PIO_PIN_LED1);

// Turn LED1 on
LED1_ON;
```

It is possible to take the principles further and create general macros for handling all ports and pins. This was just an example of how to create well-structured, maintainable and professionally looking code.

Chapter 9 contains a description how to get started with the LPCXpresso IDE. Read this chapter and follow the guide how to import the projects. Start working with project "lab 1a", which is the base for this first experiment.

After compiling and linking without errors, follow the guide how to download and run the project.

In embedded programming it is important to have full control over the variables, more specifically the number range they can hold. The original C standard was a little vague on the number of bits different variable types have. It is specified as "at least X number of bits" and there is a specified order between different types. However in embedded programming the exact number of bits is important to keep track of. Therefore it is common to have an include file that have created/specified new variable types with the number of bits exactly specified. We will use this setup in all experiments.

Include a file called **type.h** in all program files. The main content of the file is presented below:

```
#if defined ( GNUC )
#include <stdint.h>
#else
/* exact-width signed integer types */
typedef signed char int8_t;
typedef signed short int int16_t;
typedef signed int int32_t;
typedef signed __int64 int64_t;

/* exact-width unsigned integer types */
typedef unsigned char uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int uint32_t;
typedef unsigned __int64 uint64_t;
#endif // __GNUC__

#ifndef NULL
#define NULL ((void *)0)
#endif

#ifndef FALSE
#define FALSE (0)
#endif

#ifndef TRUE
#define TRUE (1)
#endif
```

As seen, there are four signed and four unsigned variable types of length 1, 2, 4 or 8 bytes (8, 16, 32, or 64 bits). The file also declares the commonly used constants: NULL, FALSE and TRUE.

Code becomes much more portable (between different compilers) if a common include file like this is used. It also becomes more readable.

7.2.2 Lab 1b: GPIO and Bit Masking

There is hardware support in the GPIO peripheral block for accessing selected bits, as opposed to accessing all of them. This is described in the LPC111x user's manual, chapter 12.4.1 – Write/read data operations. In short, the GPIOxDATA register can be accessed on many different addresses. The address used to access the register determines which bit(s) that is/are accessed.

Below is a copy of a function from NXP's driver library for the LPC111x. As seen, it is a general function for manipulating any GPIO output (any port, any pin). The array named MASKED_ACCESS[...] is used to get the correct access address to the GPIOxDATA register, given which bit(s) to access. Note that the function below only allows one bit at a time to be accessed. (1<<bitPosi) is used to index into array MASKED_ACCESS[...]. It is possible to create more general access functions where several pins can be controlled simultaneous, for example MASKED_ACCESS[(1<<bitPosi1) | (1<<bitPosi2) | (1<<bitPosi3)].

```
*****
** Function name: GPIOSetValue
**
** Descriptions: Set/clear a bitvalue in a specific bit position
**               in GPIO portX(X is the port number.)
**
** parameters: port num, bit position, bit value
** Returned value: None
**
*****/
void
GPIOSetValue( uint32_t portNum, uint32_t bitPosi, uint32_t bitVal )
{
    // Check that bitVal is a binary value - 0 or 1
    if (bitVal < 2 )
    {
        /* The MASKED_ACCESS registers give the ability to write to a specific bit
```

```

* (or bits) within the GPIO data register. See the LPC11/13 user manual
* for more details.
*
* (1<<bitPosi) gives us the MASKED_ACCESS register specific to the bit
* that is being requested to be set or cleared.
*
* (bitVal<<bitPosi) will be either be 0 or will contain a 1 in the
* appropriate bit position that matches the MASKED_ACCESS register
* being written to.
*/
switch ( portNum )
{
    case PORT0:
        LPC_GPIO0->MASKED_ACCESS[(1<<bitPosi)] = (bitVal<<bitPosi);
        break;
    case PORT1:
        LPC_GPIO1->MASKED_ACCESS[(1<<bitPosi)] = (bitVal<<bitPosi);
        break;
    case PORT2:
        LPC_GPIO2->MASKED_ACCESS[(1<<bitPosi)] = (bitVal<<bitPosi);
        break;
    case PORT3:
        LPC_GPIO3->MASKED_ACCESS[(1<<bitPosi)] = (bitVal<<bitPosi);
        break;
    default:
        break;
}
}
}

```

Create a similar, general function for setting the direction of any GPIO pin (input or output). Call this new function `GPIOSetDir`. The function's input parameters shall be port number, bit number and direction.

After that, recreate the program from the previous experiment using these two new functions.

It is good programming practice to place functions that are related in separate files. It will enhance the source code structure and make it easier to maintain and understand in general. An accompanying include file (h-file) declares the functions that are exposed to other source code files.

Place the GPIO related functions in a separate file called `gpio.c` and create an include file, `gpio.h`, that declares the exposed functions.

Also, in order to keep the file `main.c` reasonable short move all defines that are related to the board to a separate include file `board.h`.

7.2.3 Lab 1c: Delay Function – LED Flashing

Next, design a program that flash with the LED – 50 ms (milli seconds) on, 150 ms off, 50 ms on and finally and 750 ms off. Continuously repeat this 1000 ms cycle.

It is a common task in embedded systems to operate on exact time and control external devices exactly. In this case a LED.

One obvious solution is to create a delay function. An example is listed below that forces the CPU to execute NOP (no operation) instructions in a loop. Use this function and test different values in order to establish a relationship between the number of NOPs and the actual delay in time.

```

/*
 * Delay by executing a given number of NOPs.
 */
void
delayNops(uint32_t nops)
{
    volatile uint32_t i;

    for (i = 0; i < nops; i++)
        asm volatile ("nop");
}

```

About how many NOPs are needed for a 1 second delay? _____

What does this tells you about the execution speed of the LPC111x? _____

Note that delay loops like this should never be used in real programs. All processor execution time is "lost" in the loop and no other useful work is done. Also, the delay can vary depending on what other parts of the system do (for example how much time is spent in interrupt routines, which will be introduced later on). Later on we will explore other method of creating exact delay functions, so for now, the loop method will have to do.

Create a function that delays execution a specified number of milliseconds (as input parameter). Place the function in a separate file `delay.c`. After that, create the program that double-flash the LED according to the specification above.

Now can be a good time to get acquainted a bit more with the debugger, specifically single stepping. This means that the debugger let the microcontroller execute one statement at a time, and stops after every line. Note that for this to work the compiler optimization must not be turned on too heavily. `-O0` and `-O1` is typically what work. More optimization will rearrange the code so there are no clear boundaries between the source code statements (= rows in the source code).

Instead of pressing the **Start/Resume** button it is possible to press the **Step Over** or **Step Into** buttons. Both "step" buttons will stop execution after the current statement. The difference is that if the statement involves a function call, *Step Over* will not single step through all statements in the function that is called. *Step Into* will do just this.

The current experiment exemplified just perfect where the different is. When hitting the delay function it is best to *Step Over*, instead of into it. Single stepping through all the loop iterations would take forever.

Add a loop counter in the forever loop. Set a breakpoint in the forever loop in `main()` so that execution halts every loop iteration. Verify that it is possible to get the value of the loop variable by hovering over the variable. Remove the breakpoint and test single stepping, with both *Step Over* and *Step Into*.

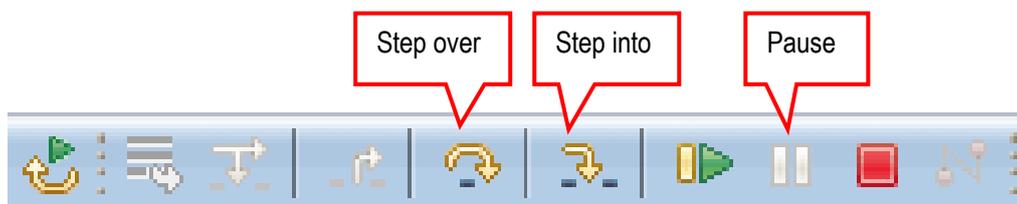


Figure 10 – LPCXpresso IDE Step Over/Into Buttons

7.2.4 Lab 1d: Morse Code

Create a function that flashes the LED according to the Morse code alphabet. Check the Wiki for details: http://en.wikipedia.org/wiki/Morse_code. The function shall take an arbitrary string as input and send the string by flashing the LED.

7.3 Read a Digital Input

In this experiment you will learn how to control the I/O pins of the LPC111x as inputs. More specifically you will learn how to read a digital input that reflects that state of a push-button.

7.3.1 Lab 2a: Read Push-button

We will start with reading the state of push-button SW2 in the schematic, which is found in the schematic on page 4, lower left corner. SW2 is connected to signal GPIO_17-KEY. Figure 11 illustrates where SW2 can be found in the schematic. On schematic page 2, we can see that this signal is connected to PIO1_5 on the LPCXpresso LPC111x board. Figure 12 illustrates where to find the signal and also where to find the GND pin.

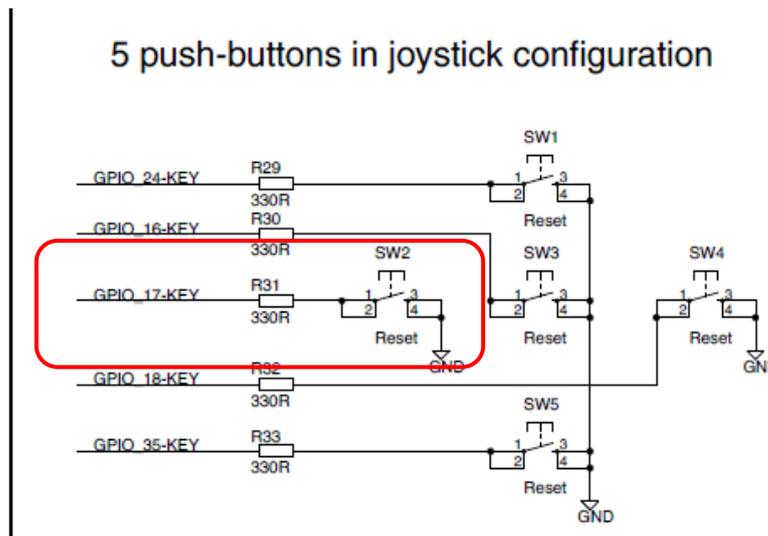


Figure 11 – SW2 on Schematic Page 4

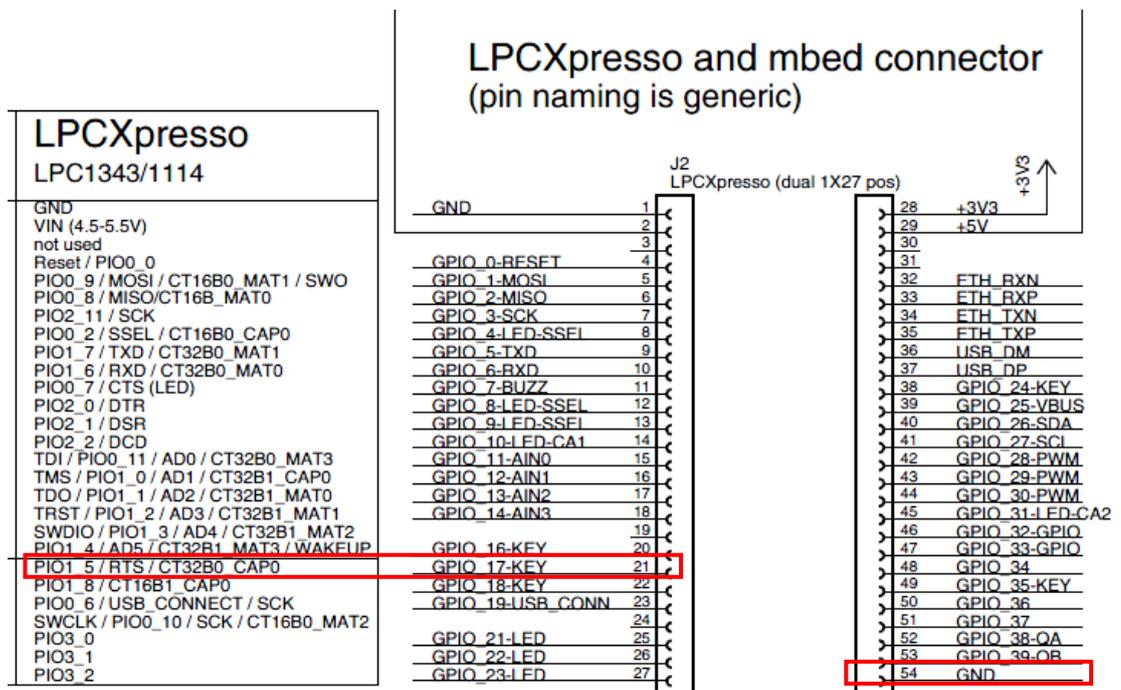


Figure 12 – Signal GPIO_17-KEY on Schematic Page 2

Keep the previously mounted LED. Get a push-button (representing SW2) and a 330 ohm resistor (representing R31) from the components bag. Note that there are two types of push-buttons; for pcb mounting and for breadboard mounting. It is the latter that shall be used now. Mount the push-button and resistor on the breadboard and connect to the LPCXpresso LPC111x board, as illustrated in Figure 13.

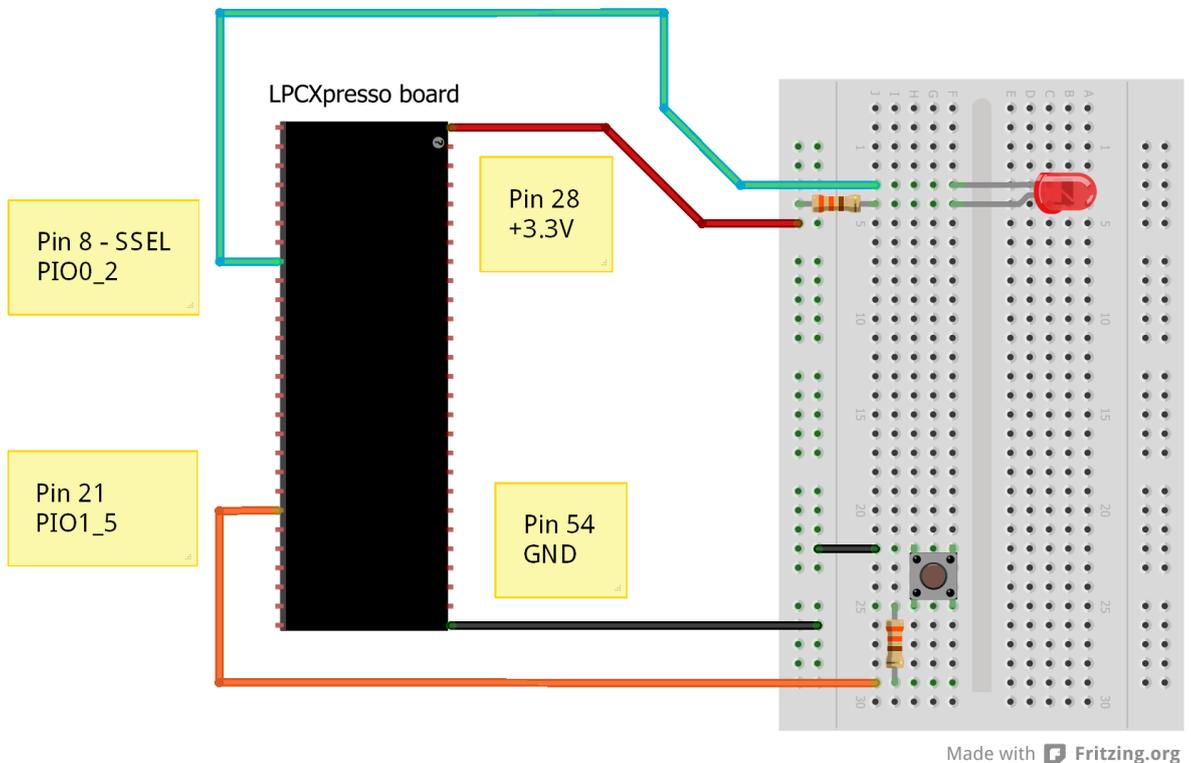


Figure 13 – Breadboard Connections for SW2 and LED

It is common that microcontroller input pins have built-in pull-up resistors. If the input is not driven the input is high. Sometimes the behavior of the pins is very programmable, for example if pull-up or pull-down resistors and input hysteresis shall be enabled. In this experiment a pull-up resistor must be enabled on the input pin. When pressing the push-button it will actively pull the input pin to ground. Else the internal pull-up resistor will pull the input high.

It is important to check the datasheet how strong the pull-up resistors are so that the external signal can pull the pin low and vice versa that the built-in pull-up resistor can pull an inactive signal high.

The series resistor is for protection if the (supposedly) input pin is an output. If that output is pulled high by the microcontroller and the push-button is pressed, the output could be damaged due to excessive current flowing to ground if a series resistor does not limit the current. The situation is not an imaginary situation. Suppose there already is an application running on the microcontroller from a previous experiment. That application might very well use the pin as an output. Before the correct application has been downloaded the damaged can happen. Therefore it is a good practice to add series resistors to all signals that can drive a microcontroller pin - the key in this case, which can drive the signal low.

Pin PIO1_5 is controlled by register IOCON_PIO1_5 (check chapter 7 - *LPC1100/LPC1100C/LPC1100L series: I/O configuration* in the LPC111x user's manual). In the description for this register we can see that there are three alternative pin functions:

- PIO1_5, a general purpose input/output, port #1, pin #5

- RTS, a control output signal for peripheral block UART
- CT32B0_CAP0, an input signal to 32-bit timer #0

By default, after reset, the register is initialized to PIO1_5, have a pull-up resistor enabled and disabled input hysteresis. As we know from the previous experiment, there is another register that controls the direction of the general purpose digital input/output and this register initialize PIO1_5 to be an input after reset.

Hence, after a reset, PIO1_5 is an input with pull-up resistor enabled. The pin is pulled high weakly which is exactly what we need. When pressing the push-button the pin will be pulled low. The input will be read high when no push-button is pressed and low when it is pressed.

In Experiment 1b, a function called `GPIOSetDir` was created. Even though the direction of PIO1_5 is correct from reset it is good programming practice to initialize the pin according to need. It is simpler for other programmers to read and understand an application if there are no hidden assumptions.

Register `LPC_GPIO1->DATA` holds the current state of the pins in port #1. Bit 5 in this register reflects the state of pin PIO1_5. Since the register reflects all pins in the port the bit of interest must be masked out. Use the same principle as presented in Lab 1a, i.e., AND with $(1 \ll \text{bitNumber})$.

Create a program that reads the state of the pin (and hence the push-button) and copy the result to a LED. Turn on the LED when the push-button is pressed. Below is the skeleton of the program that you shall create.

```
// Create defines for simpler access of LED1
#define LED1_PORT    PORT0
#define LED1_PIN     2
#define LED_ON       0    //Low output turn LED on
#define LED_OFF      1    //High output turn LED off

// Create define for simpler access of push-button
#define SW2_PIN      5

// Initialize pins to be inputs and outputs,
// set outputs to defined states
...

uint8_t ledState;

//enter forever loop
while (1)
{
    //Check if push-button is pressed (input is low)
    if ( (LPC_GPIO1->DATA & (1 << SW2_PIN)) == 0)
        ledState = LED_ON;
    else
        ledState = LED_OFF;

    //Control LED
    GPIOSetValue( LED1_PORT, LED1_PIN, ledState);
}
```

There are many things that can be done to create macro/defines to get a better abstraction structure of the program above. First, the push-button states (pressed, not pressed) can have constants defined. The `LPC_GPIO1->DATA` register can be defined as `#define SW2_DATAPORT LPC_GPIO1->DATA`. It is also possible to create a general `SW2_VALUE` macro where the pin state is returned.

Update the code above according to these principles (more general and better structured code).

It is also possible to create a general function `GPIOGetValue()`, just like `GPIOSetValue()`. This will be an exercise in the next experiment.

7.3.2 Lab 2b: GPIO and Bit Masking

As presented in Lab 1b there is hardware support in the GPIO peripheral block for accessing selected bits, as opposed to accessing all of them. This is described in the LPC111x user's manual, chapter 12.4.1 – Write/read data operations. In short, the LPC_GPIOx->DATA register can be accessed on many different addresses. The address used to access the register determines which bit(s) that is/are accessed.

The function prototype is presented below. Create a version of the function that utilizes the masked read functionality. Also create a version of the function that utilizes the bit masking we have used in previous labs.

```

/*****
** Function name:      GPIOGetValue
**
** Descriptions:     Read (bit)value in a specific bit position
**                  in GPIO portX(X is the port number.)
**
** parameters:      port num, bit position
** Returned value:   0 if bit is not set, else a non-zero value (if bit is set)
**
*****/
uint8_t GPIOGetValue( uint32_t portNum, uint32_t bitPosi)
{
    ... //implemented either with "masked read" functionality in the GPIO hardware
    ... // or via direct bit masking with GPIOxDATA & (1 << bit)
}

```

Compare which functions is fastest. A simple method is to create a loop and call the function a million times. Turn on a LED before starting the loop and turn it off after the loop. Manually clock the time the LED is on. To get the execution time for one call, divide this LED-on-time with one million.

Place the function in file `gpio.c`.

7.3.3 Lab 2c: Logic between inputs and output

In this experiment we will introduce logic between the input (push-buttons) and the output (a LED and a buzzer). Let's begin with connecting two push-buttons, SW2 (which we already have) and SW3. According to Figure 11 and Figure 12, SW3 is connected to signal GPIO_16-KEY, which in turn is connected to PIO1_4. Figure 13 below illustrates how the connection can be done on the breadboard.

Create a program that reads the two push-buttons and turn on the LED only when both are pressed simultaneous. Then change the logic so that the LED is on if only one of the push-buttons is pressed, but not both.

The program structure will be the same as in Lab 2a and 2b, a forever loop. Read both inputs and then calculate the output value and output it.

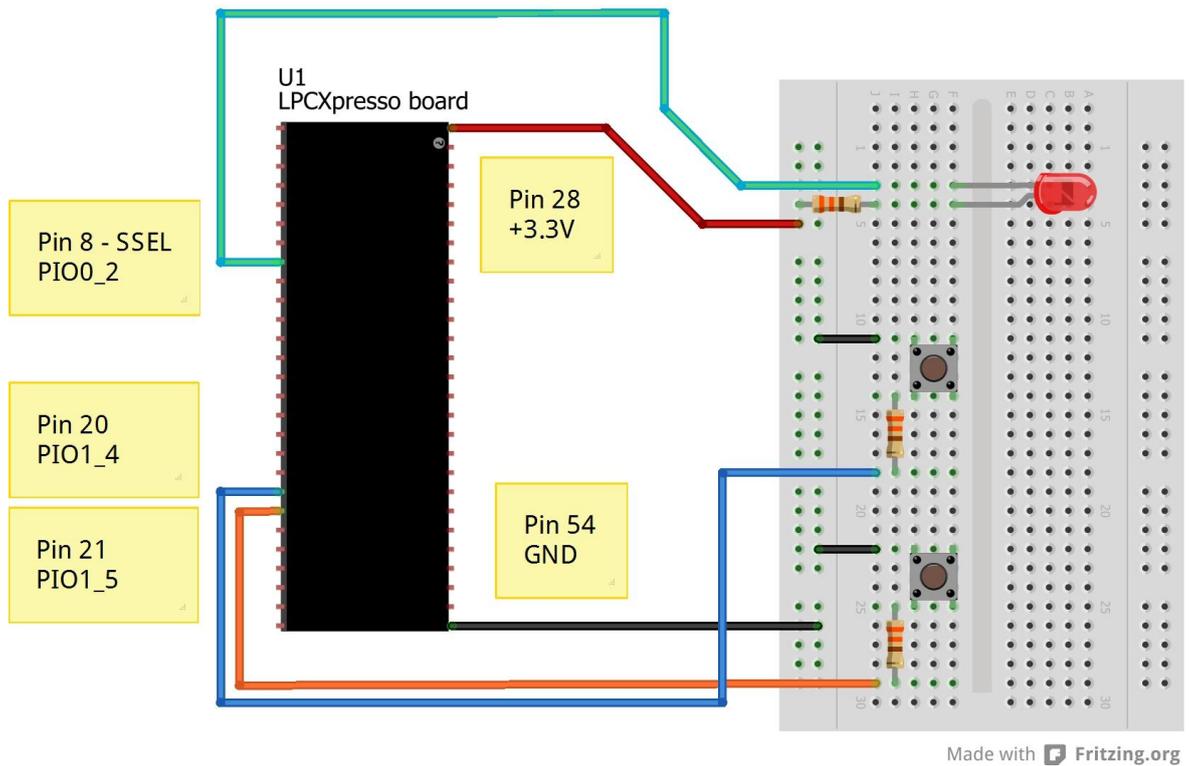


Figure 14 – Breadboard Connections for SW2, SW3 and LED

Another output device, besides a LED, is a buzzer. A buzzer outputs a single frequency tone when driving current through it. A PNP-transistor is controlling the current through the buzzer. Pulling the base of the transistor low will enable the current through the transistor (and hence the buzzer). The series resistor on the transistor's base connection limits the current (since signal GPIO_7-BUZZ will be close to ground, 0V, when pulled low by the LPC111x and a PNP bipolar junction transistor's emitter-base voltage is fixed to around 0.7V).

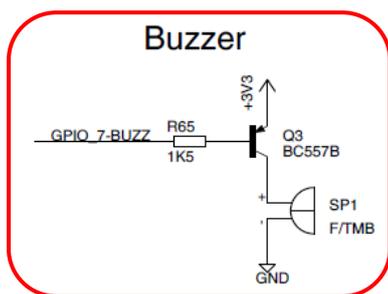


Figure 15 – Buzzer on Schematic Page 4

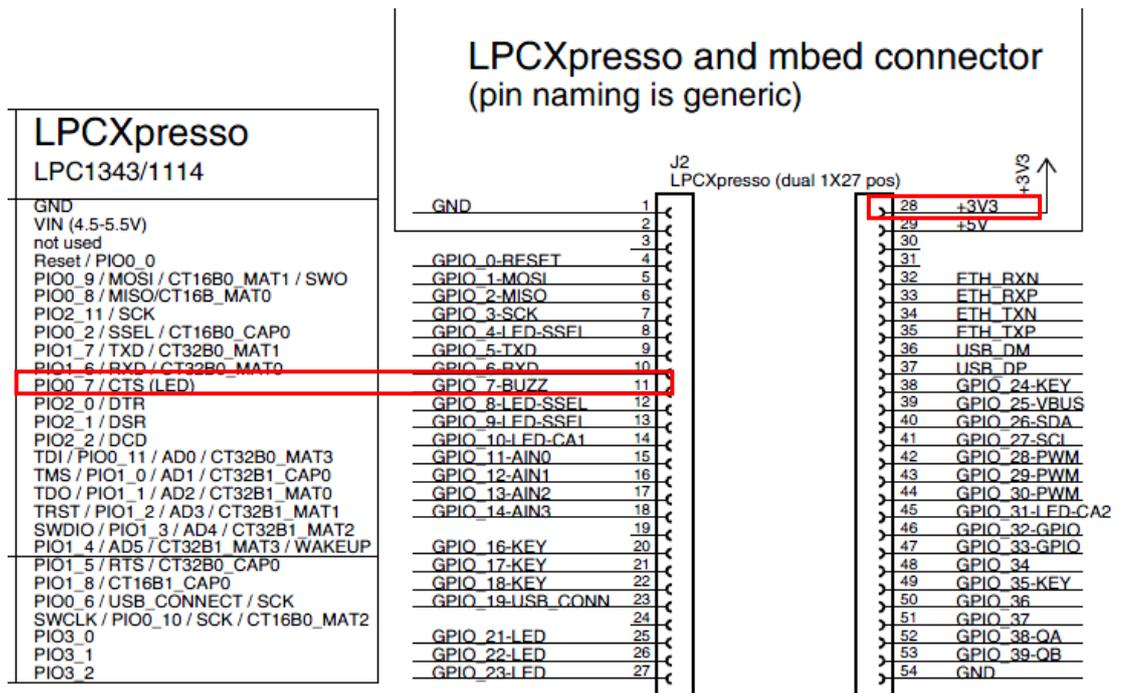


Figure 16 – Signal GPIO_7-BUZZ on Schematic Page 2

Add the buzzer to the breadboard. Some rearrangement might be needed. Note that the buzzer in the component kit might look different from the one in the picture below. Also note that both the PNP transistor and the buzzer are polarized components so it is important to turn them correct. Also note that the series resistor on the PNP base pin is 1.5 kohm (a different value than we have used so far).

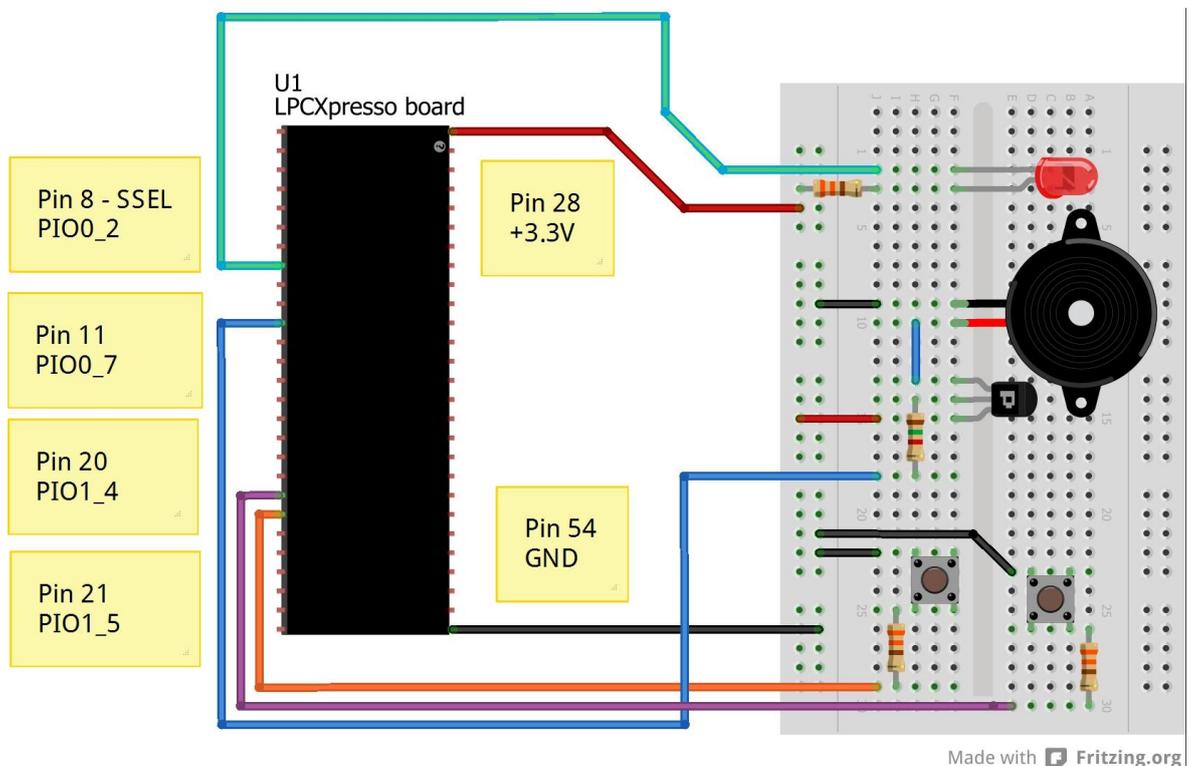


Figure 17 – Breadboard Connections for SW2, SW3, Buzzer and LED

Modify the existing code in this experiment so that the LED and the buzzer are controlled the same way (LED on = buzzer on).

As a fun extra experiment, repeat the Morse code experiment in Lab 1d with the buzzer as Morse code output instead of the LED.

Note that the buzzer will turn on during program download (on LPCXpresso boards). This is because pin PIO0_7 is also connected to a LED on the LPCXpresso board. This will drive the signal low and hence turn on the buzzer. The solution is to connect a 330 ohm (pull-up) resistor between signal PIO0_7 and +3.3V.

7.3.4 Lab 2d: Toggling LED

In this experiment we will introduce a state. Pressing the push-button shall turn the LED on. Pressing again will turn the LED off. Another way of expressing it is that the LED is toggled every time the push-button is pressed.

The structure of the program is outlined below. When the push-button first is pressed, the LED is toggled. Check the current state of the LED and inverse it. The recommended structure for this is to store the LED state in a separate variable. After having toggled the LED, the program must wait until the push-button has been released. If this last step is omitted, the LED would constantly toggle at a high rate as long as the push-button is pressed. That would not be a desirable solution since the LED can be in any state when the push-button is finally released.

```
// declare variables
uint8_t stateLED;

// Initialize pins to be inputs and outputs,
// set outputs to defined states
...

//enter forever loop
while (1)
{
    //check if push-button is pressed
    if (...)
    {
        //toggle LED
        ...

        //wait until push-button is released
        while(...);
    }
}
```

You will probably notice that the LED will toggle a little more than expected. For example when releasing the push-button, sometimes the LED will not change state. This is because of contact bounce inside the push-button. The microcontroller is so fast so it will detect multiple presses/releases. In the next experiment you will find one way of dealing with this problem.

7.3.5 Lab 2e: Sampling of Inputs

In this experiment we will introduce the concept of sampling. In the previous experiments the outputs have been controlled as quickly as possible and the inputs have been read as often as possible. Although simple, it is often desirable to have more detailed control of the system behavior.

Sampling is a concept where the state of inputs is read at defined points in time, the sample period. Outputs are also controlled/changed at these points in time. More advanced systems can have many different rates active at the same time. Some inputs are read at high rate (for example 1000 Hz, once each 1 ms) while others are read at lower date, say 10 Hz (i.e., once each 100 ms). The used rate is a trade-off between workload for the microcontroller and how fast the input can change (or how fast the

outputs must be controlled). A fast changing signal must for example be sampled often in order not to miss any important information.

In this experiment we shall sample the push-button with different sample rates. The forever-loop of the previous experiment (Lab 2d) is used. A delay function is introduced before checking push-button state. Use the delay function created in Lab 1c for this. If the delay is for example 100 ms, the effect is that the push-button is sampled at 10 Hz rate.

```
// declare variables
uint8_t stateLED;

// Initialize pins to be inputs and outputs,
// set outputs to defined states
...

//enter forever loop
while (1)
{
    //delay a specified period of time (the sample period)
    ...

    //check if push-button is pressed
    if (...)
    {
        //toggle LED
        ...

        //wait until push-button is released
        while(...);
    }
}
```

Experiment with different delay settings / sample rates and see how fast you need the sample to push-button in order not to miss a quick push-button press.

About what sample rate is needed in order not to miss any button presses? _____

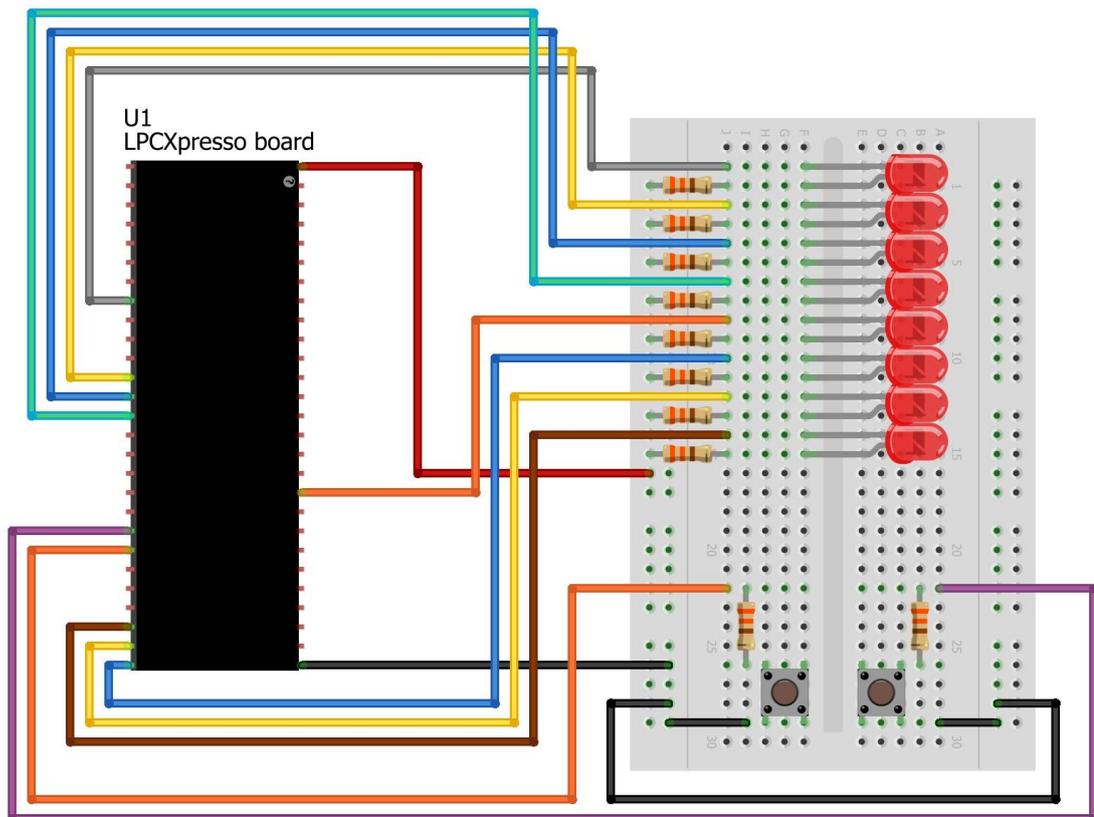
As an added bonus, the problem with contact bounce is also handled when the delay was added. That is because the microcontroller is idling in the delay-loop while the contacts bounce.

7.4 Control Multiple LEDs

In this experiment you will learn how to control multiple I/O pins simultaneously. More specifically you will learn how to control eight LEDs. This experiment builds on the knowledge you have gained from the previous experiments.

7.4.1 Lab 3a: LEDs in Running-One Pattern

As a start, create the circuit with 8 LEDs and two push-buttons as illustrated in Figure 18 below. Only one push-button is used in this experiment but in the two next, two are needed. Have a look at page 4 in the schematic to get all LED and push-button connections. LED1-LED8 and SW2-SW3 are mounted. All resistors are 330 ohm.



Made with Fritzing.org

Figure 18 – Breadboard Connections for 8 LEDs and two Push-buttons

In this experiment the 8 LEDs shall be controlled in a running-one pattern. First let the running rate be fixed. Use the delay function from previous experiments as the time base. The program structure is suggested to be as below. 1) Wait a fixed time, 2) Update the state variable (or counter, which is technically also a state) and 3) set outputs according to state.

```
//Declare variables
...

//Initialize pins to be inputs and outputs,
// set outputs to defined states
...

//Enter forever loop
while (1)
{
  //Delay a specified period of time or wait for push-button to be pressed
  ...

  //Update state/counter
  ...
}
```

```

//Update LEDs according to state/counter
...
}

```

Since there are 8 LEDs it would be suitable to define 8 states or having a counter count between 0-7 (or 1-8, if that makes more sense). The “set outputs according to state” can be discussed in more detail. One method is to first reset all outputs to their inactive state. In our case, that means setting the 8 LED outputs high (which will turn the LEDs off). After that, the state/counter determines which output to set low (turn on one LED). This structure will save code since the resetting to default state is done only once.

Another method, also outlined below, is to set all outputs to correct levels, in each state. This will duplicate much code but the program can possibly be easier to understand and maintain.

```

//Reset all outputs
...

//Set only the active output
switch(...)
{
case ...: ... break; //Set only active output
case ...: ... break; //Set only active output
case ...: ... break; //Set only active output
default:
}

or

//Set and reset the outputs
switch(...)
{
case ...: ... break; //Set only active output and reset all others
case ...: ... break; //Set only active output and reset all others
case ...: ... break; //Set only active output and reset all others
default:
}

```

It is also possible to experiment with other patterns than the running-one. For example, having 3 LEDs on simultaneous.

When the fixed running rate is working, adjust the program so that a push-button press is advancing the running LEDs instead of time. Simply replace the delay function with a while-loop that waits for a push-button press.

A little twist on above is to add a timeout functionality. If the push-button is not pressed for 5 seconds, the running LEDs should be advanced one step. How to implement that?

Tip: sample the push-button at a fixed (known) rate. Count how many times sampling is done. If too many samples without detecting a press, then a timeout has occurred. Define the timeout with a constant (`#define ...`).

7.4.2 Lab 3b: Control of Running-One Pattern

The experiment can only be done partially on the breadboard. The goal is to generalize the program from Lab 3a. Use two push-buttons for increasing and decreasing the LED running-one speed. Use two more push-buttons for controlling the direction of the LED running-one pattern, and finally use one push-button for start/stopping the LED flashing.

All-in-all, five push-buttons are needed for this. Only two are available for mounting on a breadboard. Develop the program in steps. First develop the variable speed solution. Then set the speed to a fixed value and continue developing the direction control. Then fix the direction and develop the start/stop

control. After these three steps all functionality has been developed. Use the breadboard setup as illustrated in Figure 18 above.

There are five pcb-mounted push-buttons that can be used. These push-buttons are mounted in a "joystick" structure so the up/down buttons can for example control the speed. The right/left buttons can be used to control the direction and the middle button can control start/stop.

7.4.3 Lab 3c: Rotary Switch Control of Running-One Pattern

Note that this experiment can only be done in full on the pcb since the rotary switch cannot be mounted on the breadboard. However, it is possible to simulate a rotary encoder with two push-buttons so the experiment can still be done on the breadboard, if wanted.

In this experiment, the rotation switch controls the LEDs running-one pattern. Turning the switch one step to the left shall advance the LED state on step to the left. Turning the switch one step to the right shall advance the LED state on step to the right.

The rotary switch used can also be called a quadrature rotary encoder. The encoder is named SW6 in the schematic and can be found on schematic page 5, see Figure 19 below.

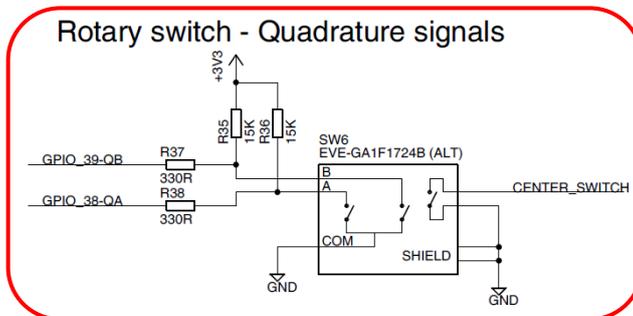


Figure 19 – Quadrature Encoder (SW6) on Schematic page 5

The encoder outputs two signals, A and B, according to Figure 20 below. The two signals vary over four states (A,B): (0,0) (1,0) (1,1) (0,1). Depending on rotation direction the four states are traversed from left to right or right to left.

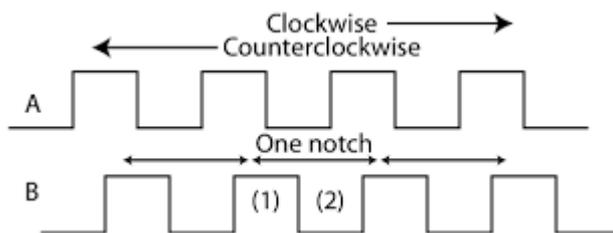


Figure 20 – Quadrature Encoder Output Signals

A rotation step can be detected by sampling the two inputs and determining what the new state is. If this new state is same as the current one, no rotation has occurred. However if there is a difference, then a rotation has been detected. If for example the current state is (A,B)=(1,0) and the new state is (1,1) the rotation is in the right direction. If the new state is (0,0) the direction is instead to the left.

Another method to detect rotations is to detect negative edges on signal A. The level on signal B (high or low) at this point in time determines the direction. Position (2) in Figure 20 above represents the counterclockwise direction (B is low) and (1) represents the clockwise direction (B is high).

How to detect a negative edge on signal A?

Tip: When sampling input A, compare with previous sample. If old sample is high and new sample is low then a negative edge has occurred.

7.5 Print Messages

So far the microcontroller has had limited possibilities to communicate with the user. Technically it would have been possible to communicate information via the Morse code experiments (via a LED or a buzzer) but it is not a very user friendly method and it would take time to communicate longer messages.

In this experiment you will learn how to print messages from the program in the LPCXpresso IDE. No breadboard work is needed for these experiments. The LPCXpresso IDE has support for something called *Semihosting*. It is a term from ARM that indicates that part of the functionality is carried out by the host. The *host* in this case is the PC, i.e., the LPCXpresso IDE. It is a very useful debug tool for small systems that do not have a dedicated communication channel for outputting debug information.

It is very easy to enable *Semihosting* in a project. Figure 21 below indicates the project setting that is needed to be carried out. Basically it is an instruction to use a special C runtime library. A library that directs printf()-output to the LPCXpresso IDE.

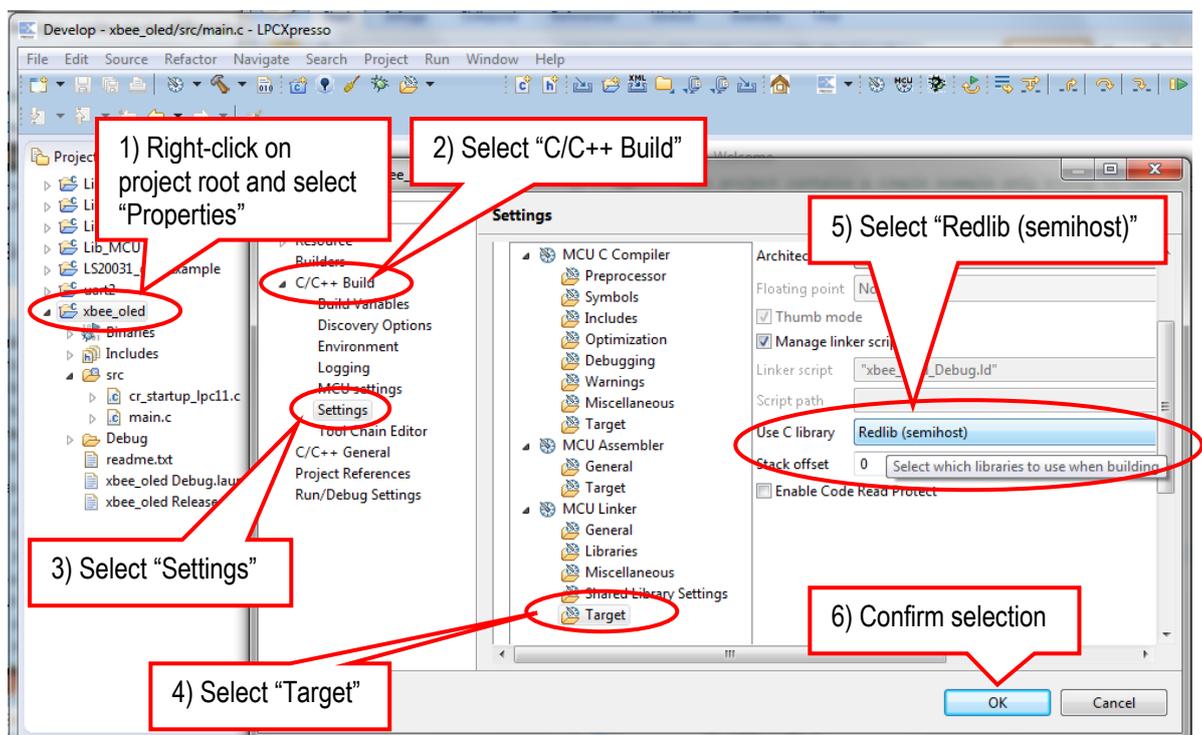


Figure 21 – Selecting Semihosting C Library

7.5.1 Lab 4a: Semihosting and printf()

In this experiment you will learn how to print messages from the LPC111x microcontroller to the console window in the LPCXpresso IDE. The code below outlines what is needed in order to use printf(). Figure 22 below illustrates how the console window looks like when executing this code.

```
//Include needed libraries
#include<stdio.h>

int main(void)
{
    printf("\nThis is a first test...\n");
    printf("that semihosting and printf() works - and it does!");

    while(1)
        ;

    return 0;
}
```

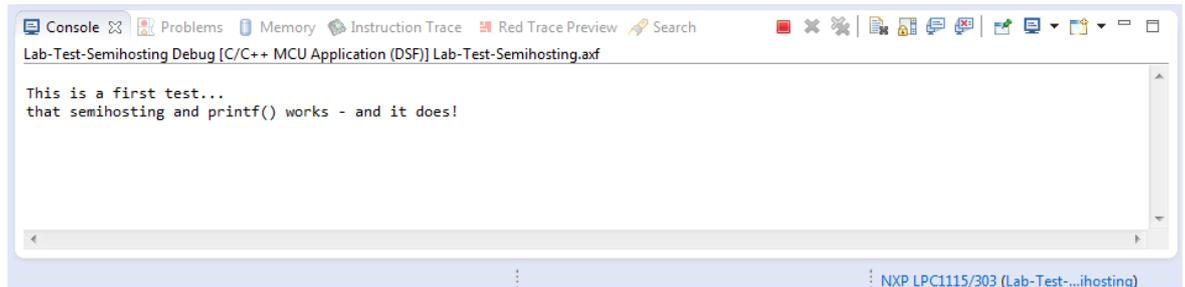


Figure 22 – Semihosting Console Output

Create a program that determines the Endianness of the microcontroller and prints the result. Assume we have a 32-bit number: 0x0AC0FFEE in hexadecimal notation. The table below illustrates how the bytes are stored differently between a big and little Endian system.

Memory address	n	n+1	n+2	n+3
Big-endian	0x0A	0xC0	0xFF	0xEE
Little-endian	0xEE	0xFF	0xC0	0x0A

Now think of a solution how to test this.

Tip: Create an unsigned int-pointer and an unsigned char-pointer. Let these pointers point to the same unsigned int-variable. Write a value in the unsigned int-variable with the unsigned int-pointer. Then read out the four parts via the unsigned char-pointer.

What Endian does the LPC111x has (little or big endian)? _____

The printf() function works like normal. It is possible to output strings and general expressions. Verify that this works.

Note that Semihosting affects code execution performance severely. Every time a data transfer takes place, the execution of the program stops during the transfer. The microcontroller cannot do anything useful whilst waiting on each transfer to complete. The blocking time depends on the LPCXpresso IDE (and the PC it executes on). Do not use printf() and Semihosting in time critical loops. In the next lab, a performance test will be carried out to get a feeling for the transfer rate.

Adding printf()-functionality to a small embedded system is a trade-off between flexibility and code size. The full implementation of printf() is very large, especially if floating point is supported. NXP has created an application note that covers basic techniques to reduce the size of code. Many things are covered along with a reduced size printf()-implementation that is supported in the LPCXpresso IDE. The application note is named: **AN10963: Reducing code size for LPC11XX with LPCXpresso**

7.5.2 Lab 4b: Semihosting Performance Test

In this experiment we will investigate the performance of the *Semihosting* functionality. Expand the **while(1)**-loop below to increment a loop counter and print the value if this counter every iteration in the loop. Also add a 500ms delay in the loop and verify that the counter increments two times per second (by observing the console window in the LPCXpresso IDE).

```
//Include needed libraries
#include<stdio.h>

int main(void)
{
    printf("\nThis is a performance test...\n");

    while(1)
        ;

    return 0;
}
```

Now remove the 500ms delay in the loop and check how fast loop counter increments now. It will not be very fast. This shows the bottleneck of the *Semihosting* functionality. It takes time to transfer the characters to the LPCXpresso IDE.

About how many characters can be transferred each second? _____
Note that this value will differ from PC to PC.

7.5.3 Lab 4c: Printing Events

In this experiment you shall create a program that writes in the console every time a push-button is pressed. For simplicity, use the breadboard setup in Lab 3.

7.5.4 Lab 4d: Reading from the Console

In this experiment we will learn how the microcontroller can read input from the console in the LPCXpresso IDE. The standard library function `getchar()` is demonstrated. The *Semihosting* implementation has limited functionality when it comes to reading from the console. The calls are blocking, meaning that the microcontroller will stay in the library function call until the user (on the LPCXpresso IDE side) has entered the characters and hit the enter key. This is not strictly following the ANSI-C definition of `getchar()`, where it should be a non-blocking call (i.e., return immediately even if no character was pressed by the user).

Test the code below.

```
//Include needed libraries
#include<stdio.h>

int main(void)
{
    printf("\nThis is a test of getchar()\n");

    while(1)
    {
        int8_t rxChar;

        rxChar = getchar();
        printf("%c", rxChar);
    }

    return 0;
}
```

Run the program and enter five characters and then hit enter. What happens?

The reason for this is that there is a queue on the LPCXpresso side. Each time `getchar()` is called on the microcontroller side, a character is removed from the queue.

Also note that text written by the user is printed in green color and text from the target system (i.e., the LPC111x) is printed in black color.

The blocking implementation of the read functions limits the usefulness. A final application would never use this functionality. It would simply not be practical to always have the LPCXpresso IDE connected to the system. It can however be a very useful functionality when debugging. The application can for example ask the user at startup if special settings shall be used. The user can then quickly test several settings without having to recompile the application.

As an extra experiment, create a program that reads input from the console and converts it to a number. Check that only digits are entered and that the final number is within the range of a 32-bit number.

7.6 Read an Analog Input

In this experiment you will learn how to convert an analog signal to a digital value. There is a 10-bit ADC (Analog to Digital Converter) on the LPC111x microcontroller. The ADC is described in *chapter 25 - LPC111x/LPC11Cx ADC* in the LPC111x user's manual. The ADC peripheral needs some initialization before it can be used. Also, the pin-muxing needs to set the analog input functionality to the pins. There are 8 inputs to the ADC and hence 8 pins that can be initialized. The function outlined below initialize the first four pins as analog inputs. The function also initializes the ADC to be ready for conversion commands. Check the LPC111x user's manual and make sure you understand the different register initializations below.

```

/*****
** Function name:          ADCInit
**
** Descriptions:          initialize ADC channel
**
** parameters:           ADC clock rate
** Returned value:       None
**
*****/
void ADCInit( uint32_t ADC_Clk )
{
    /* Disable Power down bit to the ADC block. */
    LPC_SYSCON->PDRUNCFG &= ~(0x1<<4);

    /* Enable AHB clock to the ADC. */
    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<13);

    /* Set pin-mux correct for ADC-IN0, -IN1, -IN2 and -IN3 */
    LPC_IOCON->JTAG_TDI_PIO0_11 &= ~0x8F;
    LPC_IOCON->JTAG_TDI_PIO0_11 |= 0x02;    /* ADC IN0 */
    LPC_IOCON->JTAG_TMS_PIO1_0 &= ~0x8F;
    LPC_IOCON->JTAG_TMS_PIO1_0 |= 0x02;    /* ADC IN1 */
    LPC_IOCON->JTAG_TDO_PIO1_1 &= ~0x8F;
    LPC_IOCON->JTAG_TDO_PIO1_1 |= 0x02;    /* ADC IN2 */
    LPC_IOCON->JTAG_nTRST_PIO1_2 &= ~0x8F;
    LPC_IOCON->JTAG_nTRST_PIO1_2 |= 0x02;    /* ADC IN3 */

    LPC_ADC->CR =
        ( 0x01 << 0 ) | /* SEL=1, select channel 0~7 on ADC0 */
        /* CLKDIV = Fpclk / 1000000 - 1 */
        (((SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV)/ADC_Clk-1)<<8) |
        ( 0x0 << 16 ) | /* BURST = 0, no BURST, software controlled */
        ( 0x0 << 17 ) | /* CLKS = 0, 11 clocks/10 bits */
        ( 0x0 << 24 ) | /* START = 0 A/D conversion stops */
        ( 0x0 << 27 ); /* EDGE = 0 (CAP/MAT singal falling,trigger A/D conversion) */
}

```

7.6.1 Lab 5a: Read Trimming Potentiometer

In this experiment we shall read the value of analog input #0. There are two trimming potentiometers, R7 and R20, on page 4 in the schematic. One of them, R7, is connected to GPIO_11-AIN0 and the other one, R20, is connected to GPIO_12-AIN1. These signals correspond to AIN0 and AIN1 on the ADC. The trimming potentiometers are connected to ground and +3.3V in each end. By rotating the trimming potentiometers it is possible to adjust the analog voltage to any value between 0V and +3.3V. This corresponds exactly to the input range of the ADC. 0V input gives the converted value 0 and +3.3V input gives the converted value 1023.

Figure 23 below shows the schematics around R7. The series resistor, R6 (330ohm), is just for protection in case GPIO_11-AIN0 (by mistake) becomes an output. Figure 24 shows the breadboard setup for connecting the trimming potentiometer to the LPCXpresso board, pin 15.

2 Analog Inputs

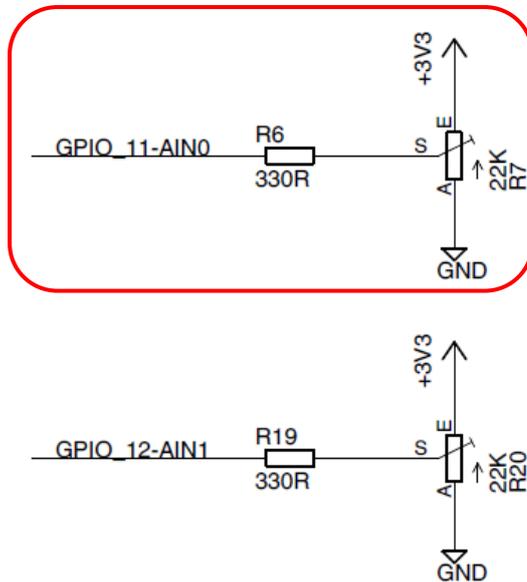


Figure 23 – Trimming Potentiometer on Schematic Page 4

Begin with building the breadboard circuit below.

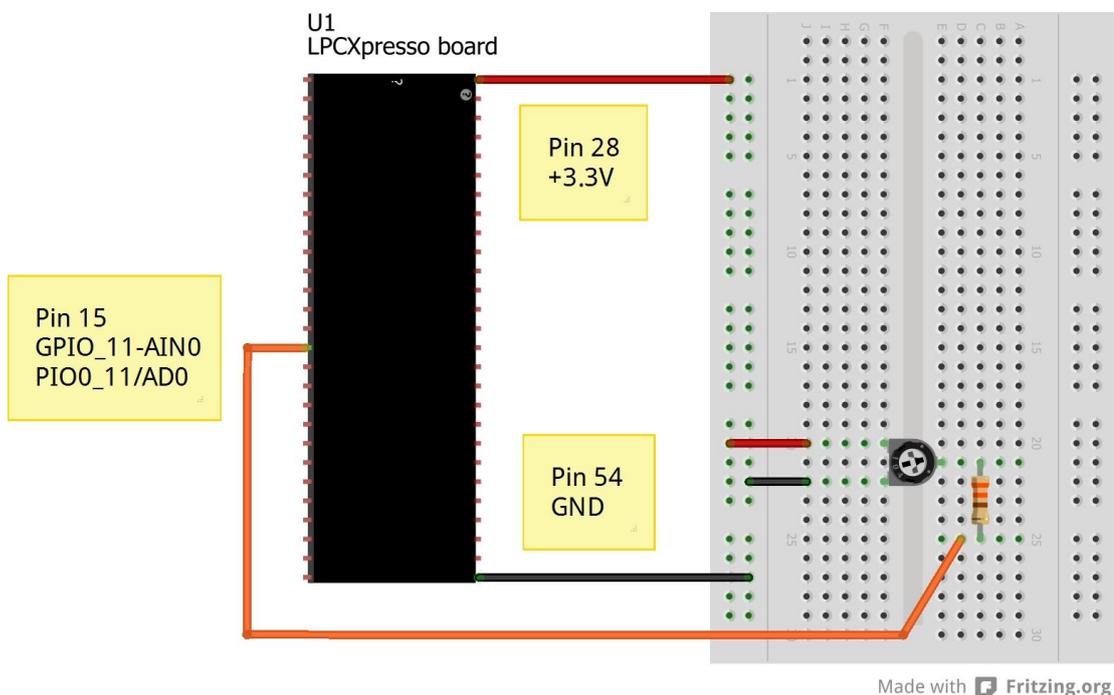


Figure 24 – Breadboard Connections for Trimming Potentiometer R7

Create a function for reading the analog value of a specified analog channel. Use constants to define the possible channels to convert.

Tip #1: Read in the ADC chapter in the LPC111x User's Manual about the CR register. By setting bit 24 and the channel to convert (in bit 0-7) a conversion is started.

Tip #2: A conversion takes some time. Check the DONE bit in register AD0GDR or AD0DRx (where x is the channel to convert).

Tip #3: After a conversion, the ADC is stopped by resetting bit 24 in the CR register.

Tip #4: When reading the converted value, note that the register value must be shifted in order to be in the interval of 0-1023 (bit 0-9 valid).

Below is the program structure to use to read the trimming potentiometer value once every 250 ms.

```
//Include needed libraries
#include<stdio.h>
...

//Define constants
#define AIN0 0
...

//Add ADC functions for initializing and reading values
...

int32_t main(void)
{
    printf("\nThis program reads AIN0 repeatedly...\n");

    //Initialize ADC peripheral and pin-mixing
    ADCInit(4500000); //4.5MHz ADC clock

    while(1)
    {
        uint16_t analogValue;

        analogValue = getADC(AIN0);
        printf("\nAIN0=%d", analogValue);

        //Delay 250ms
        ...
    }

    return 0;
}
```

You will notice some noise in the converted values. It is not always a stable value. This is quite normal to expect in a not-noise-optimized setup that we have with the breadboard and the LPCXpresso board. Besides proper hardware design, a method to handle noise is to low-pass filter the converted values. Below are two examples of how this can be done. It is a simple first-order filter. The closer to 1 ALFA is, the more filter effect is applied (the lower the cut-off frequency will be in the filter). Floating point calculations are not to recommend in smaller embedded systems. The execution time will be long for these calculations but the biggest problem is typically that the code-size will increase considerable when the C-runtime floating point libraries are linked to the program. An integer solution is to recommend instead. One example (where ALFA is 0.875) is outlined below.

```
//Floating point calculations
#define ALFA 0.95
newValue = ALFA*newValue + (1-ALFA)*newSample;

OR

//Integer calculations
newValue = ((7*newValue) + newSample) >> 3;
```

Test to filter the samples and observe that they will be more smooth and stable.

Place the ADC read functions in file `adc.c`.

7.6.2 Lab 5b: Event Threshold

Another way to handle noise (varying values from an analog signal) is to introduce threshold handling. In this exercise you shall implement a program that reports when the value of an analog signal has changed more than a set limit. Create a program that prints the value of AIN0 in the console whenever the change in value is large than 2% of the full scale.

Tip #1: 2% of 1024 steps equals 20,48, which can be rounded down to 20.

Tip #2: Remember the last reported value and compare the new sample against this value. If you compare against the previous samples value then it is theoretically possible to slowly, slowly turn the trimming potentiometer without getting any change report event.

As an extra experiment, create a program that reports changes as already implemented. However, if no changes are detected, report the current value once every 5 seconds.

7.6.3 Lab 5c: Read Light Sensor

In this experiment a light sensor will provide the analog input value (instead of a trimming potentiometer). The sensor, R24, can be found on schematic page 4, as illustrated in Figure 25 below. The more light the sensor is exposed to, the lower the resistance becomes. Adjust the program code in Lab 5a to read from AIN2 (instead of AIN0) and check what converted values to expect in different light conditions.

What is the range of values between absolute dark and full sunlight? _____

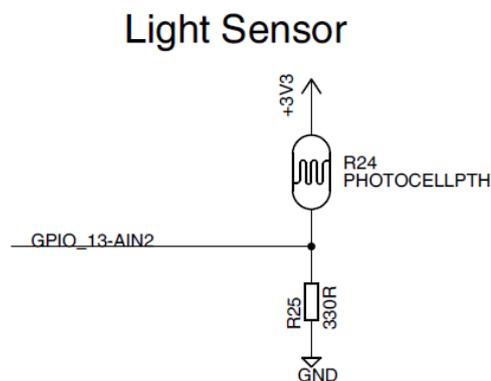


Figure 25 – Light Sensor on Schematic Page 4

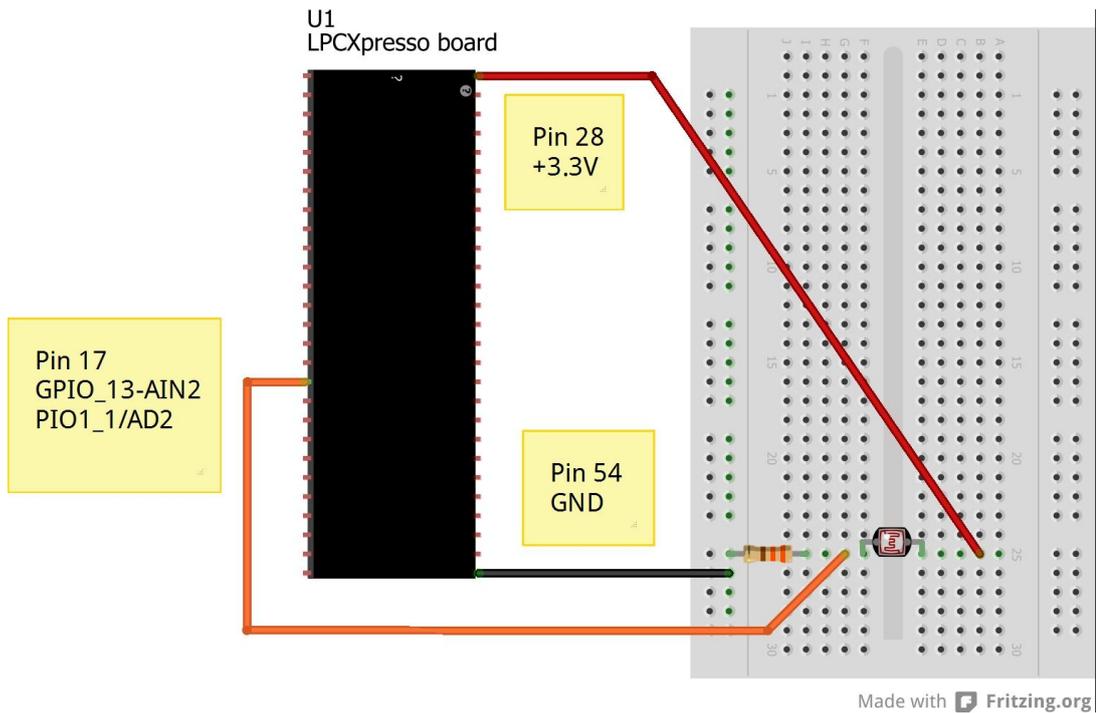


Figure 26 – Breadboard Connections for Light Sensor R24

7.6.4 Lab 5d: ADC Noise Test

As seen and experimenting with in Lab 5a and 5b, here is noise in converted analog values. In this experiment we shall investigate the ADC noise in more detail. We shall gather statistical information about the noise distribution.

Create an application that gathers 1 000 000 samples from AIN0 and sort these values according to frequency in occurrence. One solution is to create an array of 1023 32-bit variables and let each of these variables be a counter representing the number of times that particular value has been observed. This solution would require 4kByte of RAM, which is not a problem. If the resolution would have been higher, for example 12 bits then the memory consumption might be too high. An alternative solution would be to have a smaller array. Take one sample to determine in which range the values seems to be. Then set a range of for example ± 32 around this value. Also have two special counters that represent values under or above the edge values. This way it would be simple to check if the range accidentally is bad.

The result shall also be printed in the console in a user-friendly way, so that the result is easy to understand for the user.

What can the possible source of noise be? _____

possible by increasing the number of iterations but the trade-off is lower frequency of the duty cycle. This may, or may not, be a problem as we will investigate in later experiments.

```
//Set wanted duty cycle
wantedDutyCycle = ...

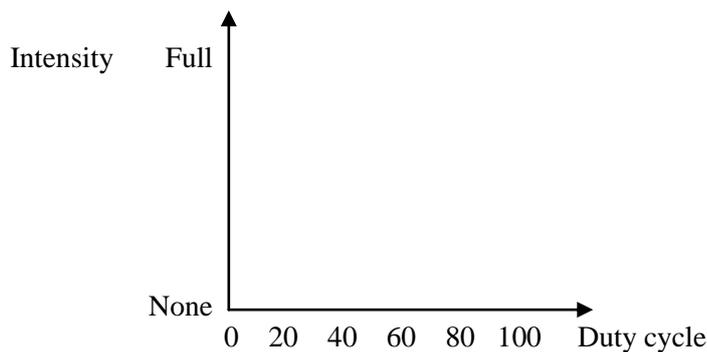
//Set output high
...

//Enter duty cycle generating loop
for (loopCounter=0; loopCounter<100; loopCounter++)
{
  if (loopCounter == wantedDutyCycle)
    //Set output low
    ...
}
```

Create a program that generate a fixed duty cycle and controls a LED. Define the duty cycle with a constant in the program.

About what frequency does the PWM signal have? _____
(measure with an oscilloscope or logic analyzer, if possible)

Draw a diagram of the perceived LED intensity with different duty cycles:



7.7.2 Lab 6b: PWM Control of a LED, cont. 1

In this experiment let the value from the trimming potentiometer control the duty cycle. Build on the code that was developed in Lab 6a. Read the AIN0 value and set duty cycle between 0-100% accordingly.

As an extra experiment, let the light sensor control the duty cycle. If the room is dark have full intensity on the LED and vice versa.

7.7.3 Lab 6c: PWM Control of a LED, cont. 2

In Lab 6a, the duty cycle frequency was fixed to a relatively high value. In this experiment you shall investigate how this frequency affects the LED intensity. At some point (when lowering the frequency) you will notice a flickering on the LED.

Create a program where one trimming potentiometer controls the duty cycle and another trimming potentiometer controller the frequency.

Tip: Add a variable delay in the loop. Let the delay be 1 us, or multiples of it. At 1 us delay, a total of 100us delay will be added per cycle. This equals 10 kHz. Measure and verify that you get about this

frequency. It will be slightly lower since more than just delays are performed in the loop. If the delay is 2 us the frequency will be 5 kHz, if 3 us it will be 3.3 kHz, etc.

At what frequency does the LED flickering become apparent? _____

7.7.4 Lab 6d: PWM Control of two LEDs

In this experiment you shall create a program that controls the intensity of two LEDs with the help of two trimming potentiometers. One loop shall now control two different PWM signals. As seen it becomes more and more complicated to control multiple signals, especially if the signals have different frequencies. The microcontroller is also fully occupied with generating the signals. If other work is performed, the PWM signals will be affected (not correct duty cycle or frequency). This is why it is typically simpler to let a timer generate the PWM signal, without software intervention (other than setup). You will investigate this in more detail later on.

7.8 Control an RGB-LED

In this experiment you will learn how to control an RGB-LED. Inside the package of the component there are three LEDs, one red, one green and one blue. The common anode is connected to the +3.3V supply and series resistors for each LED will limit the current to suitable levels (not irritating) with matching lamination from each LED. Note that the series resistor for the red LED is 1.5 kohm and 220 ohm for the blue and green LEDs. Also note that the current levels are quite low compared to what a normal RGB-LED would have.

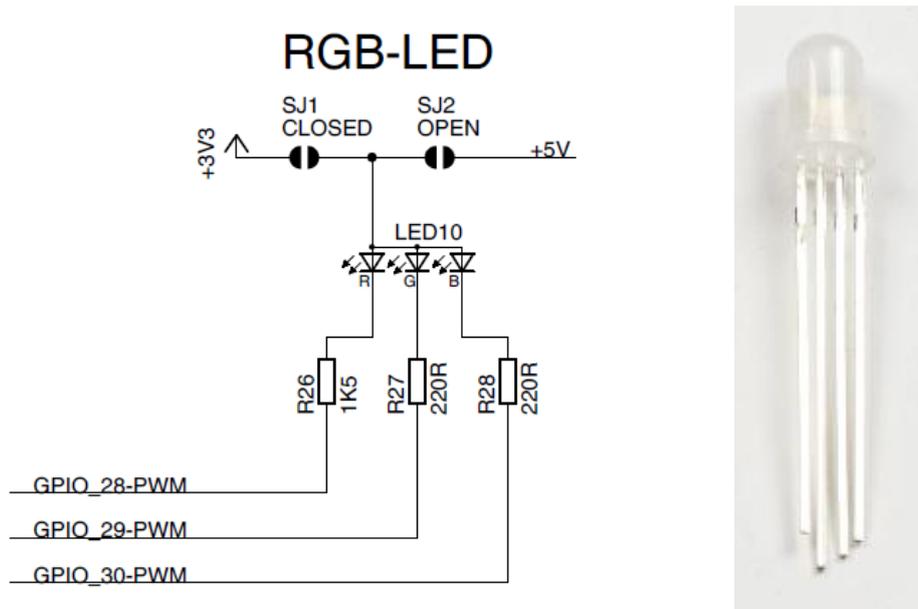


Figure 29 – RGB-LED, LED10, on Schematic Page 4

The anode is connected to +3.3V as shown in Figure 29. The pcb has shorted jumper SJ1, while SJ2 is open. There is an option to connect it to +5V instead but the currently used RGB-LED works well with +3.3V supply. Normally the problem is the blue LED, which has high forward voltage drop. Typically in the region of 3.5-4.5V. The used RGB-LED has $V_f =$ (about) 3.2V. That is also a reason why the current levels are quite low.

The RGB-LED component is also shown in Figure 29. From left to right the four pins in the picture are:

- Red-LED cathode (connected to R26)
- All LEDs anode (connected to +3.3V via SJ1, which is closed)
- Green-LED cathode (connected to R27)
- Blue-LED cathode (connected to R28)

7.8.1 Lab 7a: Test RGB-LED

In this first experiment with an RGB-LED the microcontroller will not be used. We will only use the LPCXpresso board to get the +3.3V supply. With three LEDs there are eight combinations. Verify that you can create all seven colors (besides black/dark). Insert the resistors in seven different combinations as shown in Figure 30 below.

Which colors do you get? _____

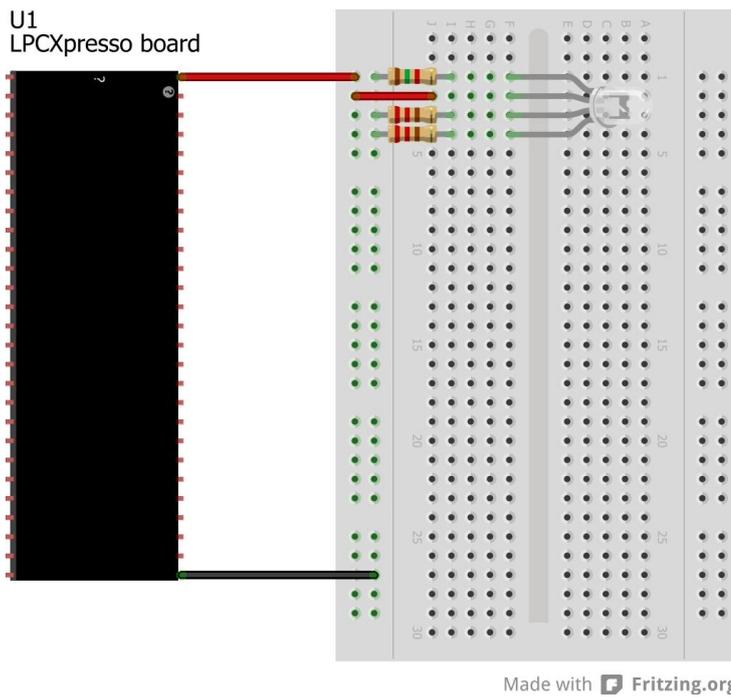


Figure 30 – Breadboard Connections for RGB-LED Testing

7.8.2 Lab 7b: Control RGB-LED

In this experiment you shall create a program that can control the intensity of each (of the three) LED. Select which color to adjust with a push-button (rotate around the three main colors, red, blue, green, at each press) and set intensity level with the trimming potentiometer.

Base the program on the knowledge developed in previous PWM-related experiments (for example Lab 6d). Below is the breadboard design that can be used.

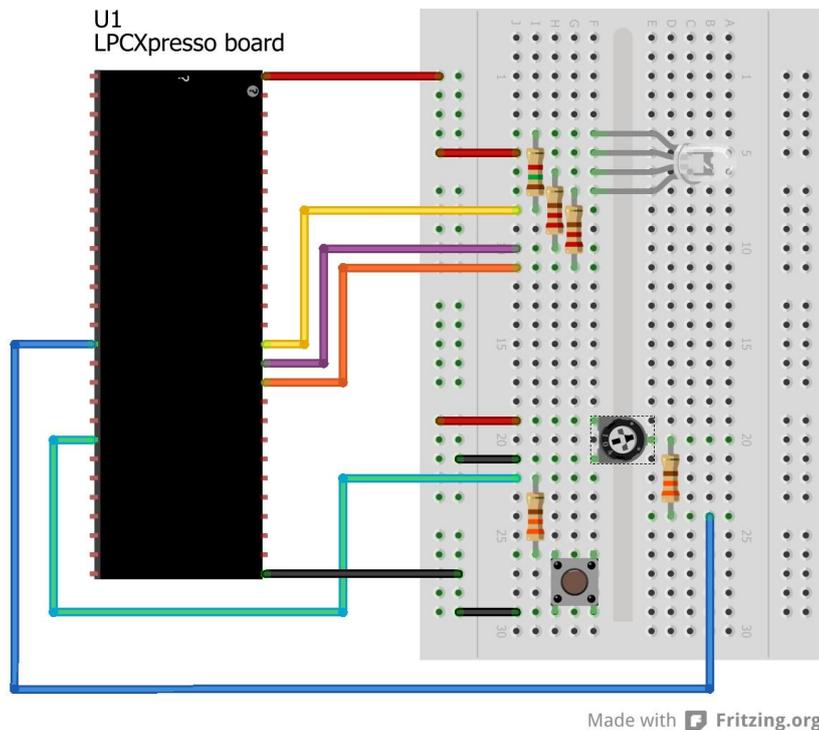
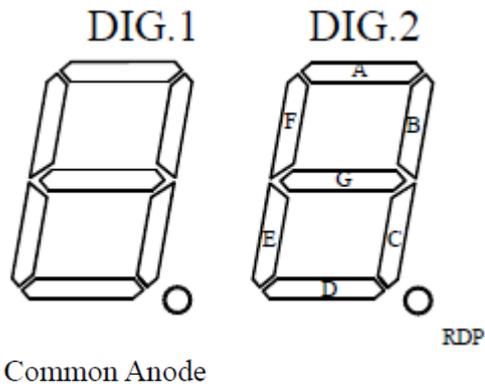


Figure 31 – Breadboard Connections for RGB-LED Experiments

7.9 Control a 7-segment Display

In this experiment you will learn how to control a 7-segment LED display. The component included in the kit actually has two 7-segment digits. More about this further on. First, let's have a look how a 7-segment display works. The name *7-segment* refers to the seven main segments, labeled A to G. See picture below. Sometimes there is also an eighth LED, a dot that is typically labeled (R)DP. It is still called a 7-segment LED even though there are actually 8 LEDs.



Common Anode

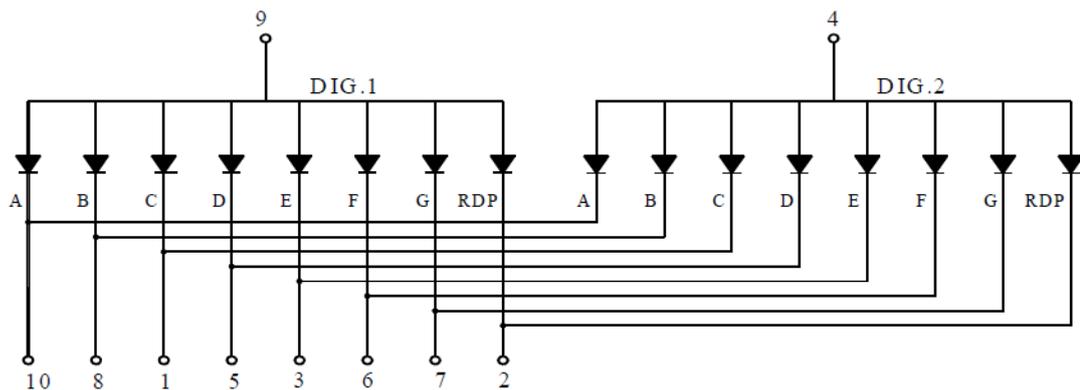


Figure 32 – Dual Digit 7-segment LED

The dual digit 7-segment LED that is used in this kit has common anodes and are multiplexed as outlined in Figure 32. With the 7 LEDs it is possible to create digits 0-9 and the six first characters in the alphabet. This makes it possible to display hexadecimal numbers. With the two digit display included in the kit it is possible to display a byte value in hexadecimal form (0x00 – 0xFF).



Figure 33 – All Hexadecimal Digits

There are other types of LED displays. See the picture below. The 7-segment display is the simplest. There are also 14- and 16-segment displays. With these it is possible to display all letters in the alphabet. There are also matrix displays in different sizes. The 5x7 formation is the smallest to get reasonable readable digits. The benefit with LED matrixes is that graphics can also be displayed.

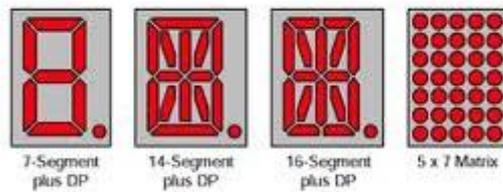


Figure 34 – Different LED Displays

7.9.1 Lab 8a: Test 7-segment Display

In this first experiment with a 7-segment display the microcontroller will not be used. We will only use the LPCXpresso board to get the +3.3V supply. Verify that you can turn on each segment of the display, by moving cables on the breadboard. The picture below illustrates the first breadboard setup with the display. Note that the picture of the dual digit 7-segment display we have is not correct in the picture below. In reality, the digits are turned 90 degrees and have two digits. There are however 10 pins on the component, just as shown below.

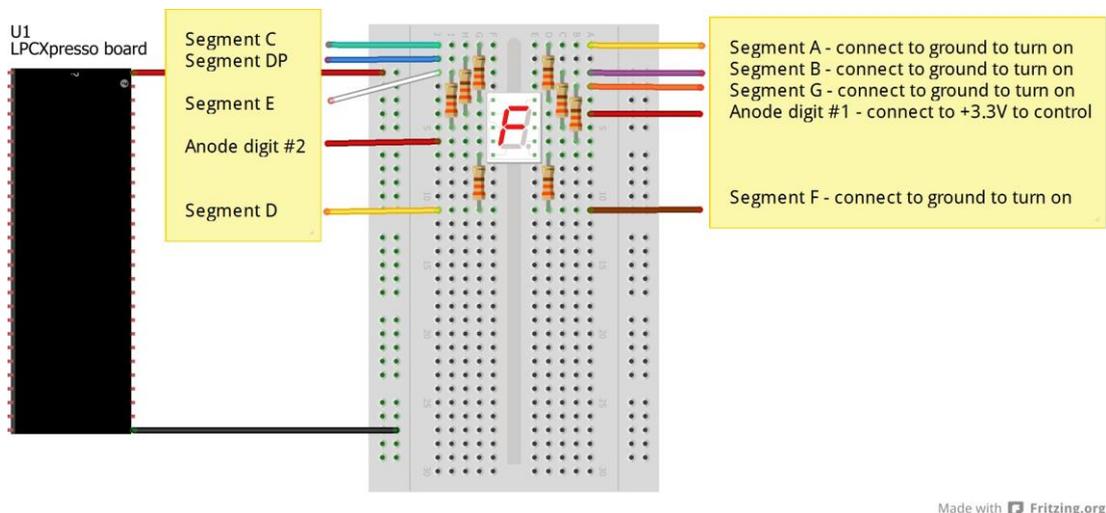


Figure 35 – Breadboard Connections for 7-segment Display Testing

Also verify that you can select with digit, of the two, to control. When working with digit #1, pin 9 shall be connected to +3.3V and when working with digit #2, pin 4 shall be connected to +3.3V. Both pin 4 and 9 shall never be connected to +3.3V at the same time. When working with both digits they must be time multiplexed. Half of the time, digit #1 is on and the other half, digit #2 is on. More about this in later experiments.

7.9.2 Lab 8b: Control 7-segment Display

In this experiment you shall control one digit of the 7-segment LED display with the microcontroller. We will use eight outputs to directly control each segment of the display. The anode of digit #1 will be connected directly to +3.3V while the anode for the second digit is left unconnected. As we have done before, the breadboard setup is prepared for the next experiments also. We will for example not use the trimming potentiometer in this experiment.

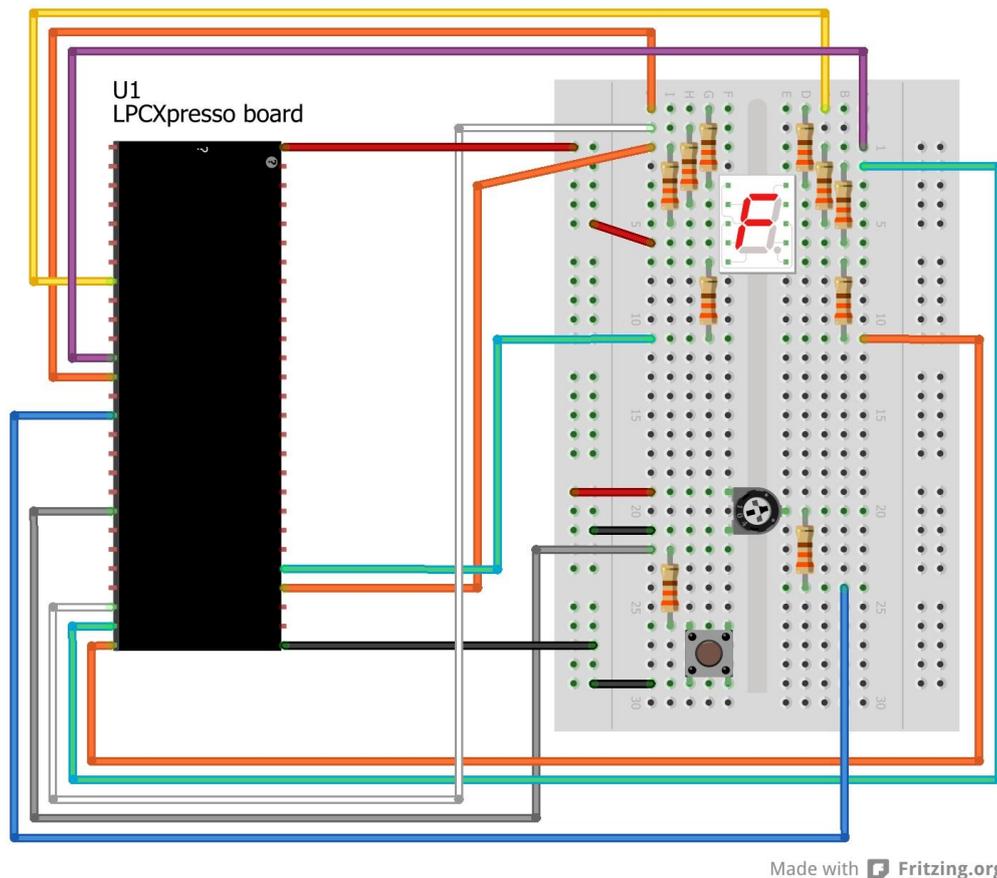


Figure 36 – Breadboard Connections for 7-segment Display

The same eight outputs are used as in the experiments with the 8 running LEDs, except for LED4 and LED5:

- LED1 (GPIO_4-LED-SSEL) corresponds to segment A
- LED2 (GPIO_8-LED-SSEL) corresponds to segment B
- LED3 (GPIO_9-LED-SSEL) corresponds to segment C
- GPIO_36 controls segment D
- GPIO_37 controls segment E
- LED6 (GPIO_23-LED) corresponds to segment F
- LED7 (GPIO_22-LED) corresponds to segment G
- LED8 (GPIO_21-LED) corresponds to segment DP

Create a program that increment a digit, 0-9 each second. It shall roll-over to 0 when 9 is reached. Let the dot LED light for 100 ms after an increment.

A suitable program structure is to create a subroutine that takes a number (0-9) as input and sets the appropriate segment outputs for each input value.

As a variation to above, modify the code so that every time you press the push-button the number is incremented.

Another variation is to create a program that creates a “running one” segment in a circular structure (segment A->B->C->D->E->F->A, etc.).

7.9.3 Lab 8c: Control 7-segment Display, cont.

In this experiment you shall present the value on ADC input #0 on one digit in the display. When turning the trimming potentiometer a value between 0 and 1023 will be read (10 bit resolution). This value shall be converted to a number between 0 and 9. One obvious conversion is $(\text{ADC value} / 1024) * 10$. In theory this conversion is correct but since we are working with integer values the term $(\text{ADC value} / 1024)$ will always be 0. By rewriting it as $(\text{ADC value} * 10) / 1024$ the precision will be kept. Assuming that the calculations are done on 16-bit variables, the calculation of $(\text{ADC value} * 10)$ is still within 16-bits of precision so no overflow will occur.

The translation $(\text{ADC value} * 10) / 1024$ works perfect for presenting the value 0-9. Consider however how a function to present the value 0-10 would have looked like. Figure 37 present the proposed conversion function (in black) and a function that would work well for the range 0-10 (in red).

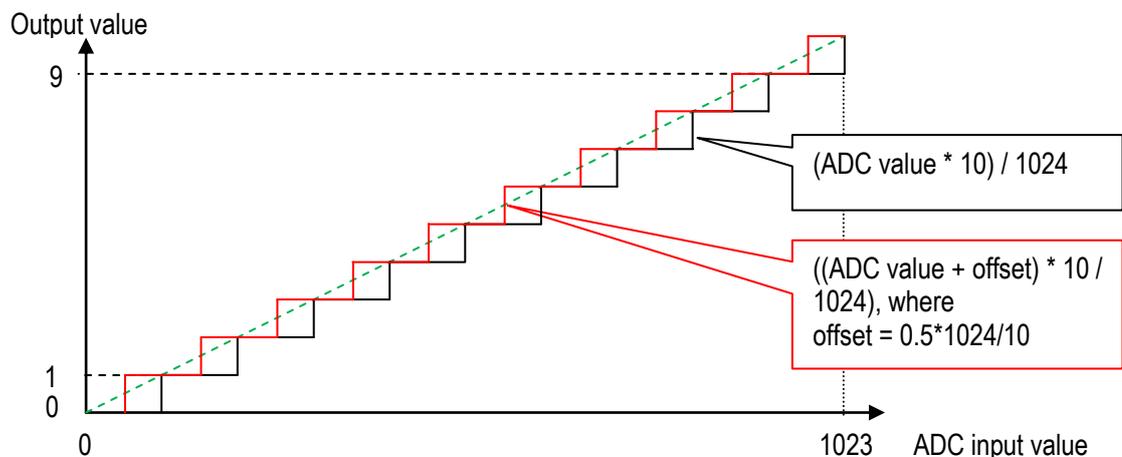


Figure 37 – Conversion Function

It would be a good program structure to place the conversion function in a separate subroutine.

7.9.4 Lab 8d: Control Dual Digit 7-segment Display

In this experiment you will create a time multiplexed control of the two digits. On the schematic there are two PNP-transistors to control the anodes of the two digits. A microcontroller output cannot supply enough current to directly drive the anodes. Therefore the PNP transistors are needed. Pulling the respective GPIO-pins connected to the base (via series resistors) low will open the transistors and hence supply current to the anodes.

In this experiment we ignore the shift register control of the segments. Instead we continue using the direct GPIO control of each segment from the previous experiments. In the following experiment the shift register will be used.

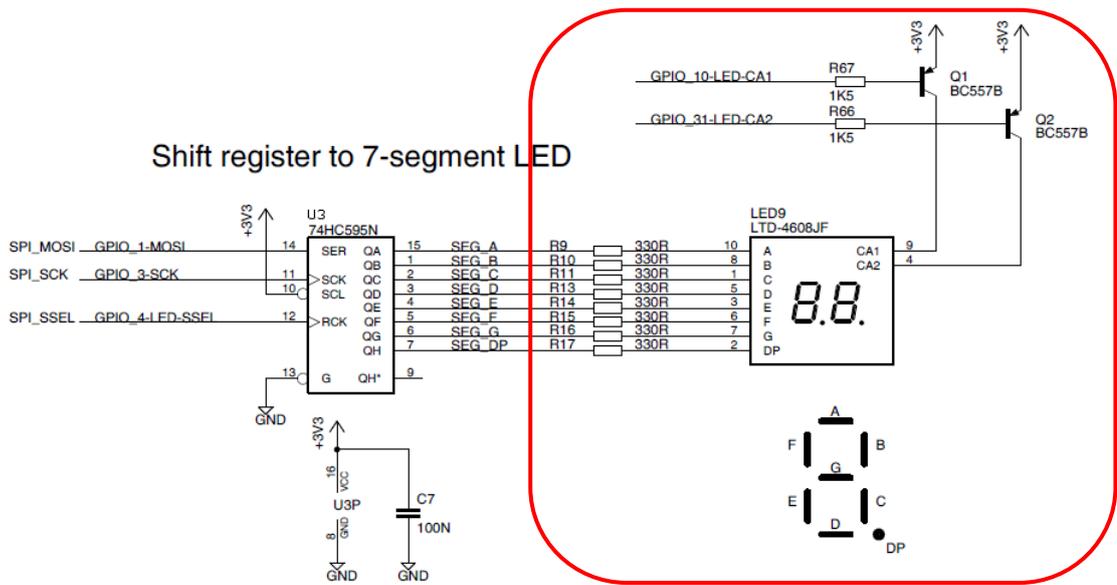
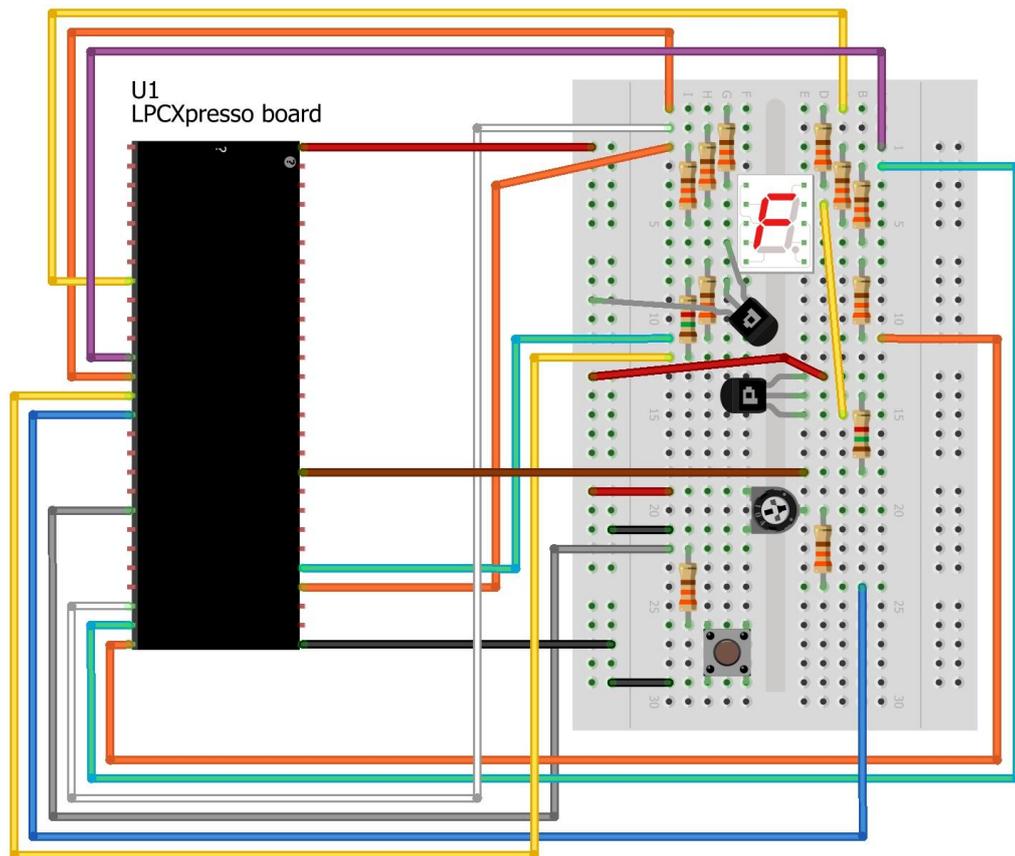


Figure 38 – 7-segment Display, LED9, on Schematic Page 4



Made with Fritzing.org

Figure 39 – Breadboard Connections for Dual Digit 7-segment Display

The suggested program structure is presented in the code block below.

```
//Time multiplexed loop for controlling two 7-segment digits
while (1)
{
  //Calculate value to present on display, e.g. read ADC input #0
  ...

  //Disconnect anode of digit #2 and reset segment outputs
  ...

  //Connect anode of digit #1 to +3.3V
  ...

  //Output value on digit #1 (control segment outputs)
  ...

  //Wait 5ms
  ...

  //Disconnect anode of digit #1 and reset segment outputs
  ...

  //Connect anode of digit #2 to +3.3V
  ...

  //Output value on digit #2 (control segment outputs)
  ...

  //Wait 5ms
  ...
}
```

Implement the time multiplexed control above and create a program just like on the previous experiment that presents the value of analog input #0 – not on one digit (0-9) but on two digits (0-99). Adjust the conversion function accordingly.

7.9.5 Lab 8e: Control 7-segment Display via Shift Register

In this experiment we shall use a shift register to control the LED segments. This is the circuit that is drawn in the schematic and designed on the pcb. The idea is to use a serial bus (called SPI) but in this experiment we will not use this bus. That is for a later experiment. Instead you shall emulate the serial bus with GPIO operations. Three signals shall be controlled, called SSEL, SCK and MOSI. These are connected to GPIO_4, GPIO_3 and GPIO_1, respectively. Figure 40 illustrates the timing of the signals. It is the signal MOSI that outputs the different segment values. The SCK signal clocks in the value of the MOSI signal on its rising edge. Signal SSEL shall be low during the clock-in process. When SSEL goes high, the value on the shift register is transferred to the outputs of the shift register. Check the datasheet of the shift register, 74HC595 for details about the shift register operation. Note the order of the bits on the MOSI signals. First the DP bit shall be output, and then segment G, etc. A zero will turn the segment on and a one will turn it off.

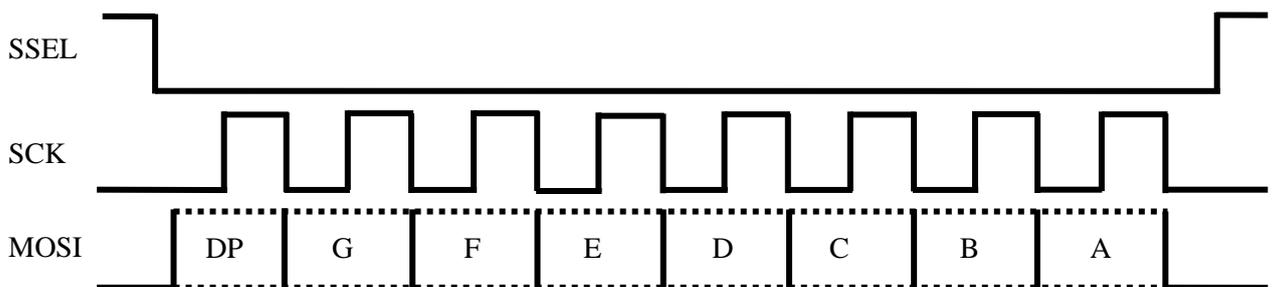


Figure 40 – SPI Shift Register Communication

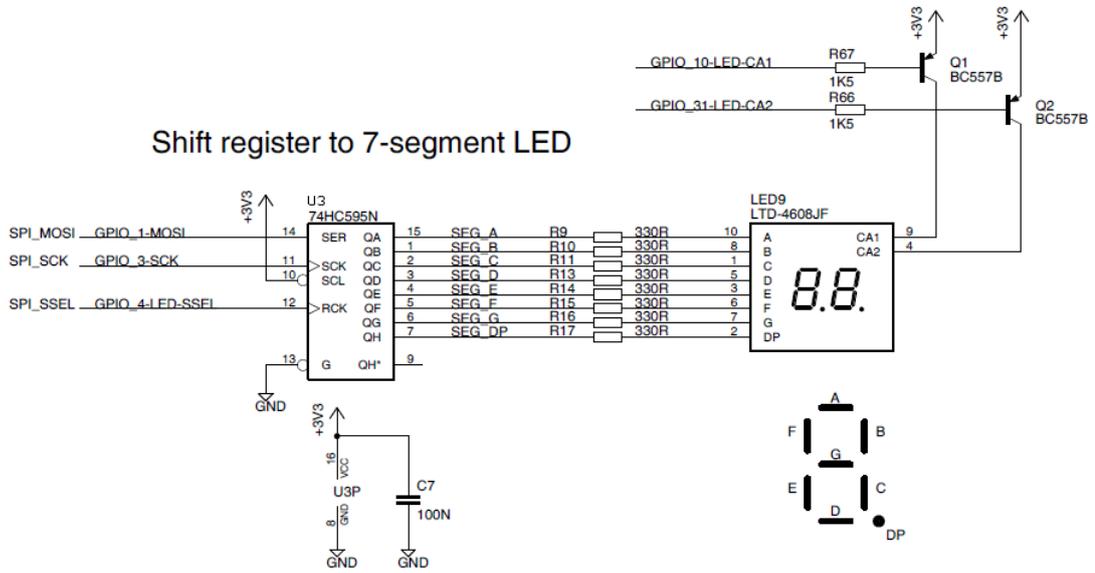
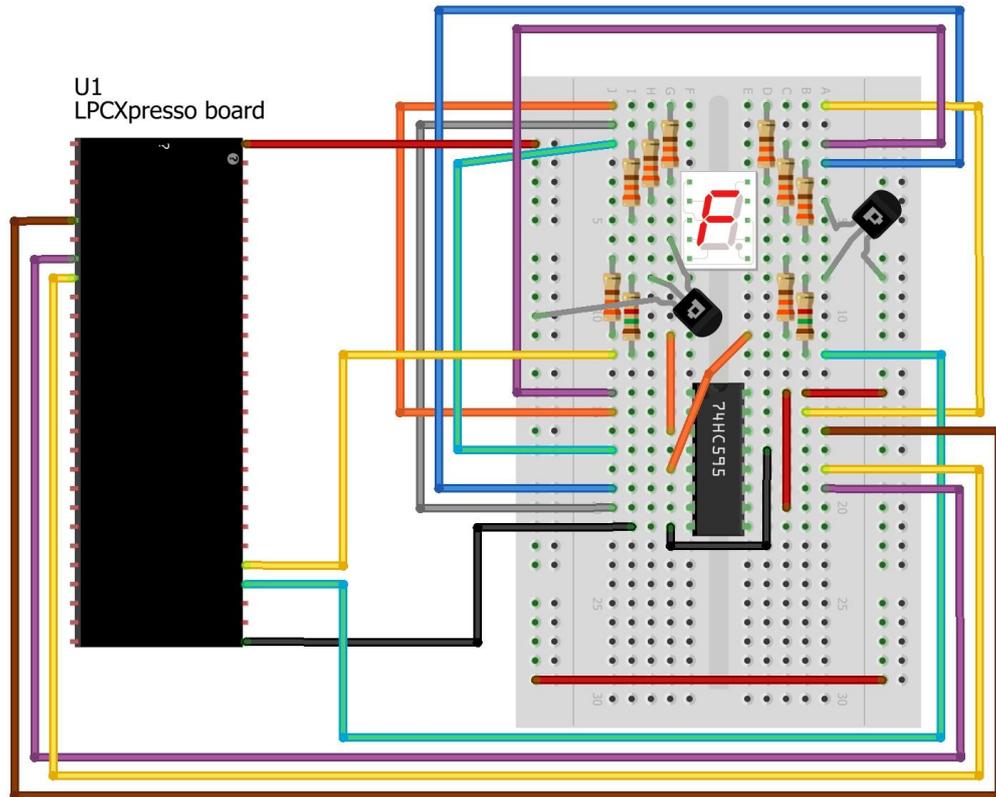


Figure 41 – 7-segment Display, LED9, with Shift Register on Schematic Page 4



Made with Fritzing.org

Figure 42 – Breadboard Connections for Dual Digit 7-segment Display with Shift Register

Create a subroutine for updating the shift register. Let the subroutine take an 8-bit variable as input, where bit 0 represents segment A, bit 1 segment B, etc. The suggested structure of the subroutine is presented in the code block below.

```
void updateShiftReg( uint8_t segments )
{
    uint8_t bitCnt;

    //Pull SCK and MOSI low, pull SSEL low
    ...

    //wait 1us
    ...

    //Loop through all eight bits
    for (bitCnt = 0; bitCnt < 8; bitCnt++)
    {
        //output MOSI value (bit 7 of "segments")
        ...

        //wait 1us
        ...

        //pull SCK high
        ...

        //wait 1us
        ...

        //pull SCK low
        ...

        //shift "segments"
        segments = segments << 1;
    }

    //Pull SSEL high
    ...
}
```

The suggested delay values are quite small (1us). They can be smaller according to the 74HC595 datasheet, but for simplicity you can use 1us. The previous delay function created had a resolution of 1ms. Update this function so that it is possible to also have smaller values in the microsecond region.

Finally create a look-up table for getting the segment values given a number between 0 and 9.

7.10 Work with a Timer

In this experiment you will learn how to work with a hardware timer.

7.10.1 Lab 9a: Create Exact Delay Function

In earlier experiments a delay loop has been used to create delays. This is not a good solution for two reasons. First, the processor will be fully occupied looping and cannot do any other useful work and it results in unnecessary power consumption. Secondly, if something interrupts the processor the delay length will no longer be correct.

In this experiment a more accurate delay function shall be created. The processor will still be idling waiting for the delay to elapse. Later on, when we explore interrupts, we will create a timer functionality that is more universal and power efficient.

Have a look in chapter 20 -32-bit counter/timer CT32B0/1 in the LPC111x user's manual for a description of the how the 32-bit timers works. Note that chapter 21 describes the same timer for the LPC1115 (the XL device family) but for our purposes the timers are identical in the LPC111x family.

A timer can be quite complicated since it can be used for creating PWM signals and capturing external events (with time stamps) and other, more or less advanced functions.

The principle to use a timer as delay function is as follows:

1. Setup and start the timer to count up from zero to a set value. The set value is calculated as: "time to delay" / "count clock period". The timer counts up/increments.
2. Wait until timer reaches the match value (a bit is then typically set in a status register).

The subroutine below implements the principles outlined above. Study the code and read in the user's manual to understand how the code works and in what way the timer is used.

```

/*****
** Function name:  delayMS
**
** Descriptions:  Start the timer delay in milliseconds until elapsed
**                32-bit timer #0 is used
**
** Parameters:    Delay value in millisecond
**
** Returned value: None
**
*****/
void delayMS(uint32_t delayInMs)
{
    //setup timer #0 for delay
    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<9); /* Enable 32-bit timer #0 clock */
    LPC_TMR32B0->TCR = 0x02;             /* reset timer */
    LPC_TMR32B0->PR = 0x00;              /* set prescaler to zero */

    //(SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV) = 48000000 => Timer clock is 48MHz
    LPC_TMR32B0->MR0 = delayInMs * ((SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV) / 1000);

    LPC_TMR32B0->IR = 0xff;              /* reset all interrupts (not needed) */
    LPC_TMR32B0->MCR = 0x04;             /* stop timer on match */
    LPC_TMR32B0->TCR = 0x01;             /* start timer */

    /* wait until delay time has elapsed */
    while (LPC_TMR32B0->TCR & 0x01);
}

```

For how long can the function above delay? _____

Create a function for microsecond delays, i.e., `delayUS()`. Let the function check so that there is no overflow in time resolution. The MR0 register has 32 bit resolution. Verify that the functions work correctly with a previous experiment, for example Lab 7b.

Place the timer related functions in file `delay.c`. This file already has delay functions.

7.11 PWM via a Timer

In previous experiment PWM signals have been generated via software. In this experiment you will learn how to work with a timer to generate PWM signals via hardware. It will free up the microcontroller for other tasks since the hardware operates without continuous software control once initialized.

In section 7.7 - Pulse Width Modulation, the principles for a PWM signal were presented. The signal has a cycle period (D, frequency = 1/D) and a duty cycle (d/D), which is the fraction of the cycle the signals is high (0-1, 0-100%).

Have a look in chapter 18 -16-bit counter/timer CT16B0/1 in the LPC111x user's manual for a description of the how the 16-bit timers works. Note that chapter 19 describes the same timer for the LPC1115 (the XL device family) but for our purposes the timers are identical in the LPC111x family.

From earlier experiments we know that a timer can be quite complicated since it can be used for many different functions. The principle to use a timer to generate a PWM signal is as follows:

1. Setup the timer to count up from zero to a match value. This value is the cycle period (D). The value is calculated as "cycle period" / "count clock period".
2. The counter counts up to this match value (cycle period value) and then restarts from zero. This repeats for as long as the timer is enabled.
3. Setup a match value, which represents the duty cycle (d/D).
4. When the cycle period counter restarts from zero the PWM output signal is set low. When the cycle period counter match the match register, the PWM output signal is set high.

As seen, there are two separate steps for creating a PWM signal. The first is to create a cycle period and the second is to create the duty cycle. The cycle period is typically fixed throughout the application execution time and is a design parameter. The duty cycle is, on the other hand, something that typically changes during the application execution time.

If the cycle period is not so critical, just "high-enough", then a suitable value for the period register can be 100. The resolution on the duty cycle is then 1% (100 steps). The match register is set to a value between 0 and 100. If higher resolution on the PWM signal is needed the cycle period can for example be set to 1000. Then the resolution is 0.1%.

In general it is no problem to have any value in the period register. The value in the match register is calculated like this (assuming 0-100% duty cycle as input parameter):

$$\text{Match register} = (\text{Cycle register value} * (100 - \text{wanted duty cycle})) / 100$$

Note the term (100- wanted duty cycle). This is because the PWM signal starts each period as low and is set when a match occurs.

We will work with 16-bit timer #1 to generate two PWM signals. The MAT0 and MAT1 signals are pinned out and will be our PWM signals. In the LPC111x user's manual, chapter 18 we find the following important sentence: "In PWM mode, three match registers on CT16B0 and two match registers on CT16B1 can be used to provide a single-edge controlled PWM output on the match output pins. It is recommended to use the match registers that are not pinned out to control the PWM cycle length." Since MAT0 and MAT1 of 16-bit timer #1 is pinned out and used as external PWM signals we select match register 2 as the cycle period register.

The subroutines below implements the principles outlined above. Study the code and read in the user's manual to understand how the code works and in what way the timer is used to generate the two PWM signals.

```

/*****
** Function name:   initPWM
**
** Descriptions:   Initialize 16-bit timer #1 for PWM generation

```

```

**
** Parameters:      cycleLength: set PWM cycle length in microseconds
**
** Returned value: None
**
*****/
void initPWM(uint16_t cycleLengthInUs)
{
    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<8); /* Enable timer #1 (enable clock to block) */

    //setup I/O pins to be MAT-outputs
    LPC_IOCON->PIO1_9  &= ~0x07;
    LPC_IOCON->PIO1_9  |= 0x01;          /* 16-bit timer#1 MAT0 */
    LPC_IOCON->PIO1_10 &= ~0x07;
    LPC_IOCON->PIO1_10 |= 0x02;          /* 16-bit timer#1 MAT1 */

    LPC_TMR16B1->TCR = 0x02;          /* reset timer */
    /* Set prescaler so that timer counts in us-steps */
    /* (SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV) = 48000000 => Timer clock is 48MHz */
    LPC_TMR16B1->PR = ((SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV) / 1000000) - 1;

    LPC_TMR16B1->MR2 = cycleLengthInUs;

    //Setup match registers to generate a PWM signal with 0% duty = constant low
    LPC_TMR16B1->MR0 = LPC_TMR16B1->MR2;
    LPC_TMR16B1->MR1 = LPC_TMR16B1->MR2;

    LPC_TMR16B1->IR  = 0xff;          /* reset all interrupts (not needed) */
    LPC_TMR16B1->MCR = (1<<7);        /* reset timer on MR2 match */
    LPC_TMR16B1->PWMC = (1<<0) | (1<<1); /* Enable PWM mode for MAT0 and MAT1 */
}

/*****
** Function name:  startPWM
**
** Descriptions:  Start 16-bit timer #1
**
** Parameters:    None
**
** Returned value: None
**
*****/
void startPWM(void)
{
    LPC_TMR16B1->TCR = 0x01; /* start timer (16B1) = start generating PWM signal(s) */
}

/*****
** Function name:  stopPWM
**
** Descriptions:  Stop 16-bit timer #1
**
** Parameters:    None
**
** Returned value: None
**
*****/
void stopPWM(void)
{
    LPC_TMR16B1->TCR = 0x00; /* stop timer (16B1) = stop generating PWM signal(s) */
}

/*****
** Function name:  updatePWM
**
** Descriptions:  Update the PWM output setting
**
** Parameters:    channel: selects with PWM signals to update (0 or 1)
**                value:   set duty cycle (a value between 0 and 100)
**
** Returned value: None
**
*****/
void updatePWM( uint8_t channel, uint8_t value)
{

```

```

uint32_t matchValue;

matchValue = (LPC_TMR16B1->MR2 * (100 - value)) / 100;
if (channel == 0)
    LPC_TMR16B1->MR0 = matchValue;
else if (channel == 1)
    LPC_TMR16B1->MR1 = matchValue;
}

```

Place the PWM related functions in file `pwm.c`.

7.11.1 Lab 10a: Control RGB-LED

In this experiment we will repeat the experiment in section 7.8 (Control an RGB-LED), specifically section Lab 7b: Control RGB-LED. Start by recreating the breadboard hardware in Figure 31 (see page 62). The red LED is controlled by signal GPIO_28-PWM, the green LED is controlled by signal GPIO_29-PWM and the blue LED is controlled by signal GPIO_30-PWM.

Signal GPIO_28-PWM can carry signal CT16B1_MAT0 (assuming that pin PIO1_9 is configured for this) and signal GPIO_29-PWM can carry signal CT16B1_MAT1 (assuming that pin PIO1_10 is configured for this). Note that signal GPIO_30-PWM (pin PIO1_11) is not connected to any timer match output. It was not possible to design the pcb to allow this. For breadboard experiments, it is however possible to for example select signal GPIO_2-MISO (pin P0_8) that can carry signal CT16B0_MAT0. Note that this is timer #0 (and not timer #1 as the code above configures).

Either you generate the third PWM signal in software (as we have done before) or you create functions for generating PWM signals from timer #0 also. In the latter case the breadboard connections must be updated (connect signal GPIO_30-PWM to pin GPIO_2-MISO).

Repeat the exact functionality in Lab 7b: Control RGB-LED, i.e., create a program that can control the intensity of each (of the three) LED. Select with color to adjust with a push-button (rotate around the three main colors, red, blue, green, at each press) and set intensity level with the trimming potentiometer.

Test to blend the three colors and see which colors it is possible to create.

7.11.2 Lab 10b: Buzzer and Melodies

In this experiment we shall use the buzzer to output tones and in the end a melody. The buzzer self-resonates with a tone of (about) 2.3kHz. What you will do is to modulate the buzzer (turn it on/off) with the frequency of the tone to produce. This tone will be audible as well as the self-resonate tone.

The code segment below illustrates how to generate a tone. Note the table with constants for producing two octaves of notes. Study the code below.

```

uint16_t notesInUs[] = {
    2272, // A - 440 Hz
    2024, // B - 494 Hz
    3816, // C - 262 Hz
    3401, // D - 294 Hz
    3030, // E - 330 Hz
    2865, // F - 349 Hz
    2551, // G - 392 Hz
    1136, // a - 880 Hz
    1012, // b - 988 Hz
    1912, // c - 523 Hz
    1703, // d - 587 Hz
    1517, // e - 659 Hz
    1432, // f - 698 Hz
    1275, // g - 784 Hz
};

/*****

```

```

** Function name:  playNote
**
** Descriptions:  Initialize 16-bit timer #1 for PWM generation
**
** Parameters:    noteInUs:   Period time (in microseconds) for tone
**                durationMs: Length of tone (in milliseconds)
**
** Returned value: None
**
*****/
void playNote(uint16_t noteInUs, uint16_t durationMs)
{
    stopPWM();
    initPWM(noteInUs);      /* Setup to generate a PWM signal with cycle time = note */
    updatePWM(0, 50);      /* Update MAT0 to generate a 50% duty cycle */
    startPWM();
    delayMS(durationMs);  /* Wait for the duration of the tone */
    updatePWM(0, 100);     /* Turn the signal off = signal constant high */
}

```

We start with a duty cycle of 50%. Half time the buffer is on and the other half it is off. It is actually possible to adjust the volume by varying the duty cycle. The shorter time the buzzer is on, the lower the volume is. It is not the duty cycle that controls the tone. It is the cycle time that controls this.

Recreate the breadboard setup from Figure 17 with one change. Connect the buzzer control to signal GPIO_28-PWM, instead of to signal GPIO_7-BUZZ. That way you can create a PWM signal with the functions that we created in the previous experiments.

Create an application that can play a song. In a song, notes can have different duration and there can be pauses between notes. Design a system where you can specify songs in a string. Then let the application decode this string and play the song.

7.11.3 Lab 10c: Control a Servo Motor

In order to complete this experiment you need an analog control servo, sometimes just called an RC servo, and also an external power supply, 4-6 volt DC (about 1 ampere). These two parts are not included in the LPCXpresso Experiment kit, but can easily be bought from electronic components distributors and RC (Radio Control) hobby suppliers.

Servos are used in many different products. The smaller ones we focus on in this experiment are found in toys. For example in small robots, rc cars, rc airplanes, etc. There is no unified color scheme for all servos for the three wires you connect to: power (+5V), ground and control signal (PWM signal). Check the datasheet of the servo that you will be using so you connect to the correct wires.



Figure 43 – Typical Servo

There are many different types and models but in this experiment we will only focus on how to control the position of the servo, which is done via a PWM signal. The cycle period can vary over a range, but 20 ms is an average value that will work on most servos. The position of the servo is controlled by the on-time of the PWM signal. 1.5 ms will place the servo in the middle/neutral position. Increasing the on-time to 2 ms will move the position to the right-most position. Decreasing the on-time to 1 ms will move the position to the left-most position. Note that the corner values can differ between servos.

Figure 45 – Breadboard with Servo Motor

7.12 Work with a Serial Bus – SPI

In this experiment you will learn how to work with the Serial Peripheral Interface Bus, or SPI bus for short. It is a synchronous bus meaning that there is an explicit clock signal. SPI builds on the master-slave concept where one unit is a master and controls the communication. The other end is the slave. Four signals are needed for communication in both directions:

- SCLK: serial clock, driven from the master
- MOSI: data signal, Master Output, Slave Input, driven from the master
- MISO: data signal, Master Input, Slave Output, driven from the slave
- SSEL or SS: Slave Select, driven from the master

Many slaves can co-exist if there are many slave select (SSEL) signals, see picture below.

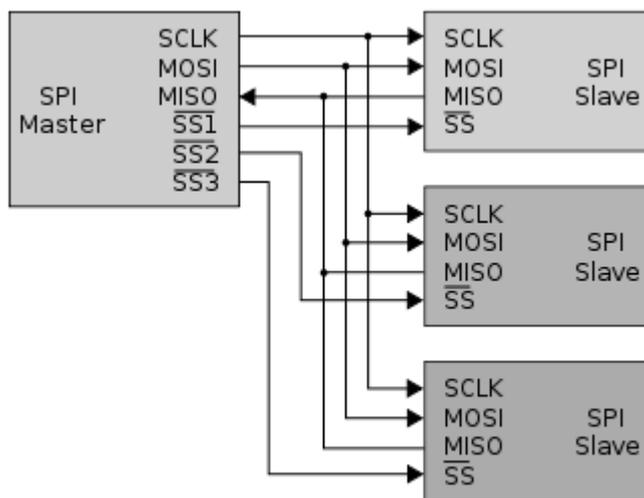


Figure 46 – SPI Master and many Slaves

The protocol defines four different modes (0-3), which have to do with which SCLK edge the data is clocked on (rising or falling) and the SCLK inactive state (high or low). Mode 0 will work fine for the SPI experiments in this section.

The master and slave connects the shift registers in a ring, see picture below. The shift registers are 8 bits long in the picture but in principle they can be other lengths also. 12-bit and 16-bit lengths are also commonly used. The most significant bit (MSB) is typically sent first on the MOSI/MISO data lines. Note that this structure results in that the master receives one byte from the slave when one byte is sent.

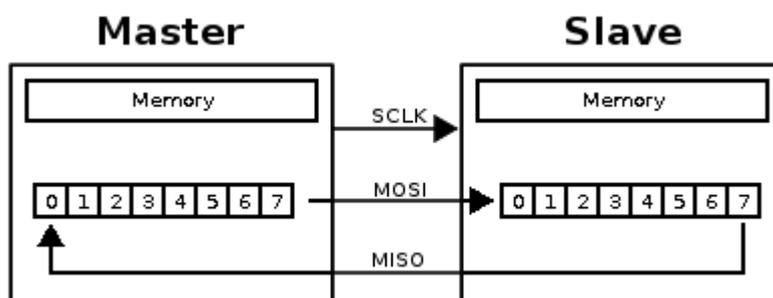


Figure 47 – SPI Master and Slave Connection

There is no specific upper frequency for the SCLK frequency. It depends on the SPI peripheral block in the microcontroller, the external SPI slave chip(s) and how far away the master and slaves(s) is/are. For breadboard experiments, the SCLK frequency should typically not exceed 1MHz. With proper pcb layout a frequency up to 20-30 MHz should not be a problem (assuming the chips involved support this frequency).

For more information about SPI, see http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus.

There are some so called decoupling capacitors that are not shown on the breadboard setups. Decoupling capacitors are added to reduce voltage dips on the supply voltage to integrated circuits. In the LPCXpresso Experiment Kit, decoupling capacitors are used on five different locations in the schematic; U3/C7, U4/C8, U5/C9, U6/C10, U7/C11. A standard value of 100nF has been selected for the capacitor.

When working on breadboard signal frequencies cannot be too high. A good rule of thumb is to keep signal frequencies below 1MHz. A breadboard is simply not a good place for high frequency electronics. The decoupling capacitors can typically be ignored on the breadboard. When soldering the components to the pcb it is however recommended to also solder the decoupling capacitors.

Have a look in chapter 14 - *LPC111x/LPC11Cxx SPI0/1 with SSP* in the LPC111x user's manual for a description of how the SSP block works. SSP (Synchronous Serial Port) is NXP's peripheral block that is capable of SPI communication (plus some other formats that will not be investigated in this experiment). It is SSP#0 that we will work with, with the following pinning:

- GPIO_1-MOSI (PIO0_9)
- GPIO_2-MISO (PIO0_8)
- GPIO_3-SCK (PIO2_11)

Further, we will use GPIO signals for SSEL. More specifically signals GPIO_4-LED-SSEL (PIO0_2) for the shift register experiments and GPIO_8-LED-SSEL (PIO2_0) for the e2prom experiment. It is possible to use the SSP#0 SSEL signal directly, which is available on PIO0_2, but in order to make the code general and supporting multiple SPI slaves we will control the SSEL signals with GPIO signals.

The code below initializes the SPI interface. Study the code below and read the LPC111x user's manual to understand the different register initialization steps. Especially note that in order to receive one byte, one byte has to be transmitted (i.e., one byte is "clocked out", one is "clocked in" at the same time).

```
#define FIFOSIZE      8

/* SSP Status register */
#define SSPSR_TFE     (1 << 0)
#define SSPSR_TNF     (1 << 1)
#define SSPSR_RNE     (1 << 2)
#define SSPSR_RFF     (1 << 3)
#define SSPSR_BSY     (1 << 4)

/* SSP CR0 register */
#define SSPCR0_DSS     (1 << 0)
#define SSPCR0_FRF     (1 << 4)
#define SSPCR0_SPO     (1 << 6)
#define SSPCR0_SPH     (1 << 7)
#define SSPCR0_SCR     (1 << 8)

/* SSP CR1 register */
#define SSPCR1_LBM     (1 << 0)
#define SSPCR1_SSE     (1 << 1)
#define SSPCR1_MS      (1 << 2)
#define SSPCR1_SOD     (1 << 3)
```

```

/*****
** Function name:    SSP0Init
**
** Descriptions:    SSP port #0 initialization routine
**                  Note that GPIO control of SSEL signal is not done, must
**                  be done separately.
**
** parameters:      None
** Returned value:  None
**
*****/
void SSP0Init( void )
{
    uint8_t i, dummy = dummy;

    LPC_SYSCON->PRESETCTRL   |= (0x1<<0);    /* Reset SSP0 block */
    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<11);    /* Enable SSP0 block */
    LPC_SYSCON->SSP0CLKDIV   = 0x02;        /* Clock to SSP0 block is divided by 2 */
                                          /* which will equal 24MHz clock rate */

    /* SSP I/O config */
    LPC_IOCON->PIO0_8        &= ~0x07;
    LPC_IOCON->PIO0_8        |= 0x01;        /* SSP0 MISO */
    LPC_IOCON->PIO0_9        &= ~0x07;
    LPC_IOCON->PIO0_9        |= 0x01;        /* SSP0 MOSI */
    LPC_IOCON->SCK_LO        = 0x01;        /* Needed to conf. PIO2_11 as SCLK */
    LPC_IOCON->PIO2_11       &= ~0x07;
    LPC_IOCON->PIO2_11       |= 0x01;        /* SSP0 SCLK */

    /* SSP0PSR clock prescale register, master mode, minimum divisor is 0x02 */
    LPC_SSP0->CPSR = 0x2;

    /* Set DSS data to 8-bit, Frame format SPI, mode #0 (CPOL = 0, CPHA = 0)
       and SCR is 7, which equals 24MHz / (CPRS*(SCR+1)) = 1500 kHz SCLK frequency */
    LPC_SSP0->CR0 = 0x0707;

    /* clear the Rx FIFO */
    for ( i = 0; i < FIFO_SIZE; i++ )
        dummy = LPC_SSP0->DR;

    /* Master mode */
    LPC_SSP0->CR1 = SSPCR1_SSE;
}

/*****
** Function name:    SSP0Send
**
** Descriptions:    Send a block of data to the SSP port, the
**                  first parameter is the buffer pointer, the 2nd
**                  parameter is the block length.
**
** parameters:      buffer pointer, and the block length
** Returned value:  None
**
*****/
void SSP0Send( uint8_t *pBuf, uint32_t length )
{
    uint32_t i;
    uint8_t dummy = dummy;

    for ( i = 0; i < length; i++ )
    {
        /* Move on only if NOT busy and TX FIFO not full. */
        while ( (LPC_SSP0->SR & (SSPSR_TNF|SSPSR_BSY)) != SSPSR_TNF )
            ;

        LPC_SSP0->DR = *pBuf;
        pBuf++;

        while ( (LPC_SSP0->SR & (SSPSR_BSY|SSPSR_RNE)) != SSPSR_RNE )
            ;

        /* Whenever a byte is written, MISO FIFO counter increments, Clear FIFO
           on MISO. Otherwise, when SSP0Receive() is called, previous data byte
           is left in the FIFO. */
    }
}

```

```

    dummy = LPC_SSP0->DR;
  }
}

/*****
** Function name:      SSP0Receive
** Descriptions:      the module will receive a block of data from
**                    the SSP, the 2nd parameter is the block length.
** parameters:        buffer pointer, and block length
** Returned value:    None
**
*****/
void SSP0Receive( uint8_t *pBuf, uint32_t length )
{
    uint32_t i;

    for ( i = 0; i < length; i++ )
    {
        /* Write dummy output byte (0xFF) to shift in new byte */
        LPC_SSP0->DR = 0xFF;

        /* Wait until the Busy bit is cleared */
        while ( (LPC_SSP0->SR & (SSPSR_BSY|SSPSR_RNE)) != SSPSR_RNE )
            ;

        *pBuf = LPC_SSP0->DR;
        pBuf++;
    }
}

```

Place the SPI related functions in file **spi.c**.

7.12.1 Lab 11a: Access Shift Register

In this experiment a shift register, the 74HC595 chip, shall be connected to the SPI bus. A byte will be transmitted to the shift register and then read back. The external shift register can be seen as a one-byte memory. Not very cost effective or high performance but in this experiment focus is on the principles and getting to know the SPI bus and the SSP peripheral block.

As a first part in this exercise, update the SSP0Init() function to take an input parameter to control the SCLK frequency. This can be done more, or less, complicated. To keep it simple, just calculate the SCR bits in the CR0 register. It will not give full coverage, but at least some range.

Create an application the send one byte (8 bits) and then reads it back. Verify that the read-back byte is correct. Use GPIO_4-LED-SSEL (PIO0_2) as SSEL signal and do not forget to initialize this GPIO signal as an output and to also control the signal levels during SPI communication. It shall be high when no communication takes place. Pull the signal low before transmitting the byte and then pull it high after the transmission.

Rebuild the breadboard circuit in Lab 8e: Control 7-segment Display via Shift Register (on page 68). The 7-segment display is not needed but as preparation for the next experiment it is simplest to rebuild it all. The breadboard setup is repeated below with one extra wire – the MISO signal.

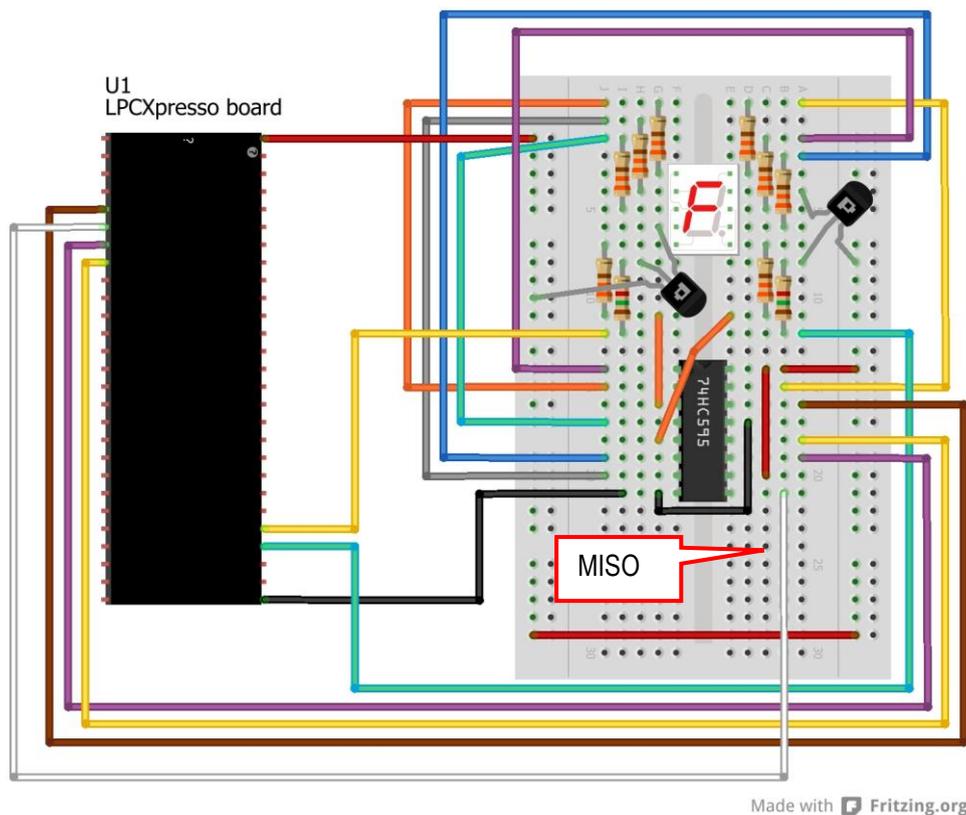


Figure 48 – Breadboard with Shift Register and 7-segment Display, with MISO

7.12.2 Lab 11b: Control 7-segment Display

In this experiment we shall revisit Lab 8e: Control 7-segment Display via Shift Register (on page 68) again. In that experiment the SPI bus was simulated in software. Now the SSP peripheral block shall be used for the SPI communication. Refresh your memory of the schematic by looking at Figure 41 again. Rebuild the breadboard circuit in Figure 42 (which you should have done already in Lab 11a). The MISO signal is no longer needed since the content of the shift register is of no interest.

Use GPIO_4-LED-SSEL (PIO0_2) as SSEL signal and do not forget to initialize this GPIO signal as an output and to also control the signal levels during SPI communication, just like in the previous Lab.

7.12.3 Lab 11c: Access SPI E2PROM

In this experiment we will interface the 25LC080 chip, which is a 1024 byte serial E2PROM that directly interfaces the SPI bus. In this experiment signal GPIO_8-LED-SSEL (PIO2_0) is used as SSEL signal.

Have a look at the datasheet for the 25LC080 chip, for example here:

<http://ww1.microchip.com/downloads/en/DeviceDoc/22151b.pdf> or search Microchip's website if the link does not work.

The 25LC080 chip has a set of instructions. All SPI transmissions begin with the instruction to execute. The needed parameters (for the instruction) are then transmitted. The set of instructions are shown in the picture below (the picture comes from the 25LC080 datasheet).

TABLE 2-1: INSTRUCTION SET

Instruction Name	Instruction Format	Description
READ	0000 0011	Read data from memory array beginning at selected address
WRITE	0000 0010	Write data to memory array beginning at selected address
WRDI	0000 0100	Reset the write enable latch (disable write operations)
WREN	0000 0110	Set the write enable latch (enable write operations)
RDSR	0000 0101	Read STATUS Register
WRSR	0000 0001	Write STATUS Register

Figure 49 –25LC080 Instruction Set

To read in the memory region, a start address (16-bit address) is transmitted after the READ instruction. In total, three bytes are transmitted from the microcontroller to the 25LC080 chip before bytes can be read from the memory. As many bytes that are of interest can be read out in the read operation. An internal address counter is incremented after each transmitted byte. If the highest address is reached (0x03FF for this chip), the address counter rolls over to address 0x0000. Note that the SSEL signal (or CS that it is called in the picture below) is low during the complete operation.

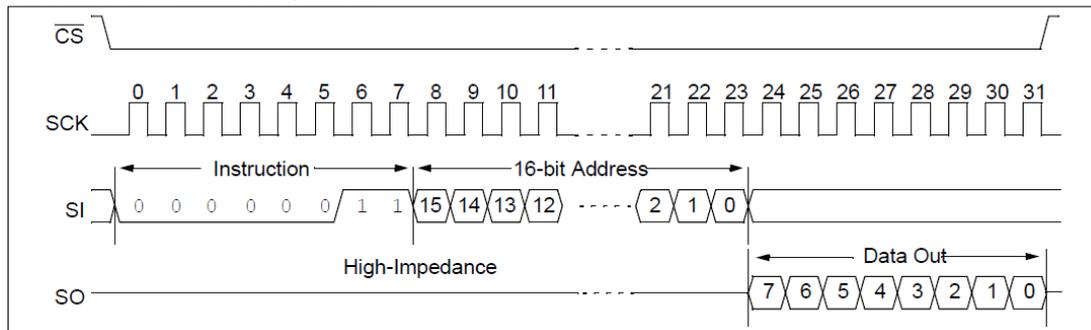
FIGURE 2-1: READ SEQUENCE

Figure 50 –25LC080 Read Sequence

To write in the memory region, the WRITE instruction is used. Similar to the read operation, a 16-bit address is transmitted to set the start address of the write operation. One or many bytes can be written at the same time, see the pictures below.

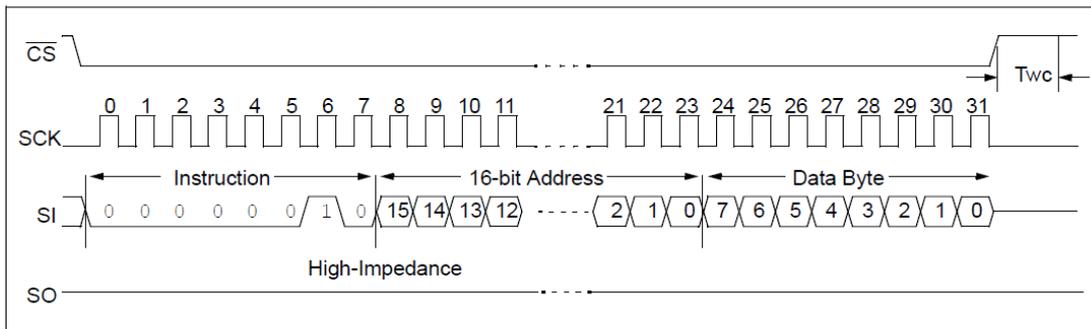
FIGURE 2-2: BYTE WRITE SEQUENCE

Figure 51 –25LC080 Byte Write Sequence

Depending on chip version (C or D, check chip package marking for details), the maximum number of bytes to write is 16 or 32. Note that all bytes must be in the same page. Physical page boundaries start at addresses that are integer multiples of the page buffer size (16 or 32 bytes). It is for example allowed to write 7 bytes from address 4 to 10. It is not allowed to write 7 bytes from address 27 to 33 since a page boundary will then be crossed (true for both 16 and 32 byte page versions).

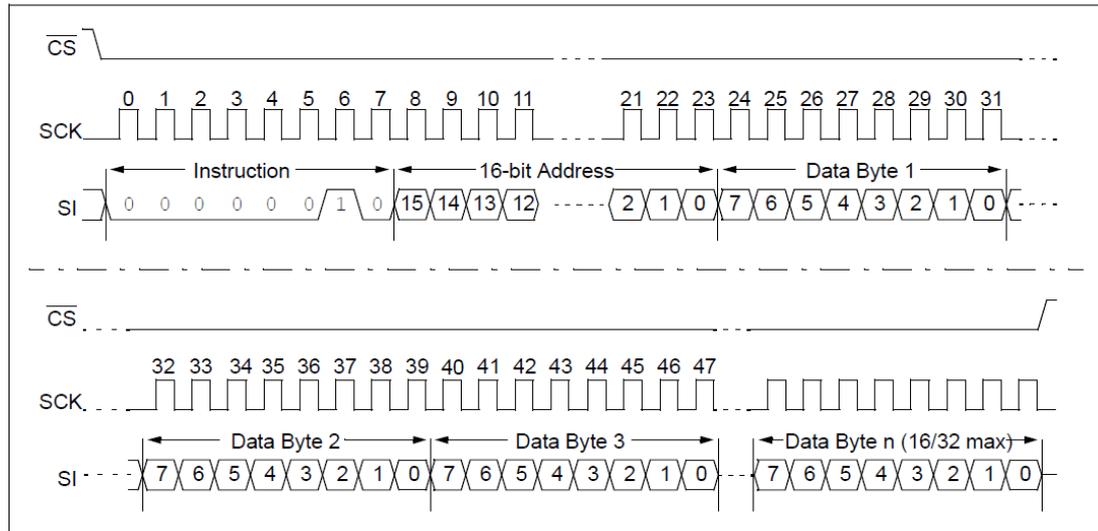
FIGURE 2-3: PAGE WRITE SEQUENCE

Figure 52 –25LC080 Page Write Sequence

The 25LC080 chip contains a write enable latch. This latch must be set before any write operations are allowed. The WREN instruction sets the latch, i.e., enable a write operation. The WRDI operation resets the latch, i.e., block write operations. Note that the write enable bit must be set before every write operations. It is automatically reset after a successful write operation.

The WREN and WRDI instructions have no parameters. They are just one-byte instructions send to the 25LC080 chip. Note that the SSEL/CS signal must be brought high after the transmissions in order for the instructions to be actually executed. The WREN instruction is shown in the picture below. The WRDI instruction is similar and not shown.

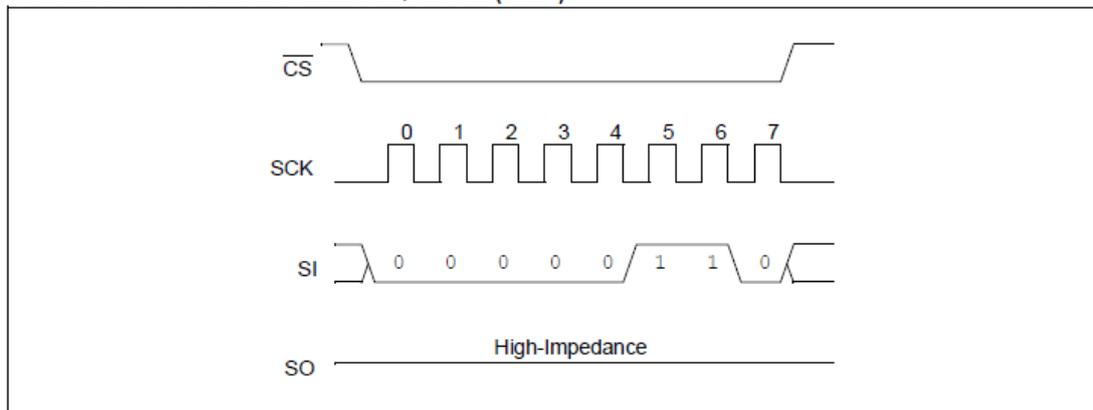
FIGURE 2-4: WRITE ENABLE SEQUENCE (WREN)

Figure 53 –25LC080 Write Enable Sequence

There is also a status register. It can be read at any time, even during a write operation. It is possible to check the status of a write operation and detect when it is ready. It is also possible to read the write enable latch state as well as controlling write protection of blocks of the memory region. Read the datasheet for details.

Study the code below. It contains an initial framework for reading and writing to the 25LC080 chip.

```

/* SPI E2PROM command set */
#define INST_WREN      0x06      /* MSB A8 is set to 0, simplifying test */
#define INST_WRDI     0x04
#define INST_RDSR     0x05
#define INST_WRSR     0x01
#define INST_READ     0x03
#define INST_WRITE    0x02

```

```

/* RDSR status bit definition */
#define RDSR_RDY      0x01
#define RDSR_WEN      0x02

#define SSEL_GPIO_8_PORT PORT2
#define SSEL_GPIO_8_PIN  0
#define SSEL_HIGH      1
#define SSEL_LOW       0

/*****
** Function name:      spiE2PROMread
** Descriptions:      This function will read bytes from the SPI E2PROM
** parameters:        address in memory region, buffer pointer and block length
** Returned value:    None
**
*****/
void spiE2PROMread( uint16_t address, uint8_t *pBuf, uint32_t length )
{
    uint8_t buf[3];

    //pull SSEL/CS low
    GPIOSetValue(SSEL_GPIO_8_PORT, SSEL_GPIO_8_PIN, SSEL_LOW);

    //output read command and address
    buf[0] = INST_READ;
    buf[1] = (address >> 8) & 0xff;
    buf[2] = address & 0xff;
    SSP0Send(&buf[0], 3);

    //read bytes from E2PROM
    SSP0Receive(pBuf, length);

    //pull SSEL/CS high
    GPIOSetValue(SSEL_GPIO_8_PORT, SSEL_GPIO_8_PIN, SSEL_HIGH);
}

/*****
** Function name:      spiE2PROMwrite
** Descriptions:      This function will write bytes to the SPI E2PROM
** parameters:        address in memory region, buffer pointer and block length
** Returned value:    None
**
*****/
void spiE2PROMwrite( uint16_t address, uint8_t *pBuf, uint32_t length )
{
    uint8_t buf[3];

    //Insert code here to break up large write operation into several
    //page write operations...
    //Do not forget to add a 5ms delay after each page write operation!

    //pull SSEL/CS low
    GPIOSetValue(SSEL_GPIO_8_PORT, SSEL_GPIO_8_PIN, SSEL_LOW);

    //output write command and address
    buf[0] = INST_WRITE;
    buf[1] = (address >> 8) & 0xff;
    buf[2] = address & 0xff;
    SSP0Send(&buf[0], 3);

    //send bytes to write E2PROM
    SSP0Send(pBuf, length);

    //pull SSEL/CS high
    GPIOSetValue(SSEL_GPIO_8_PORT, SSEL_GPIO_8_PIN, SSEL_HIGH);
}

```

Add functionality in the write operation to check so that no page boundaries are crossed. Even better, add functionality to break up a large write block to smaller, correctly addressed page writes. Do not forget to add functionality in the write function to set the write enable latch before every write operation.

Create a program that writes a string and reads it back to verify that the write operation was successful. Also let the program print the content of the memory locations directly after power-up. By doing so it is also possible to verify that the SPI E2PROM is a non-volatile memory that keeps the content over a power cycle. Build the breadboard setup below and verify that it is possible to write and read in the memory region on the 25LC080 chip. Note that the 330 ohm series resistor on the MISO pin (pin 2 of the 25LC080) is not strictly needed. It is a safety precaution in case a faulty or wrong program executes on the LPC111x, making the MISO pin an output in the LPC111x. If this happens then the signal will have two drivers. This can cause damage to the respective pin drivers on the chips.

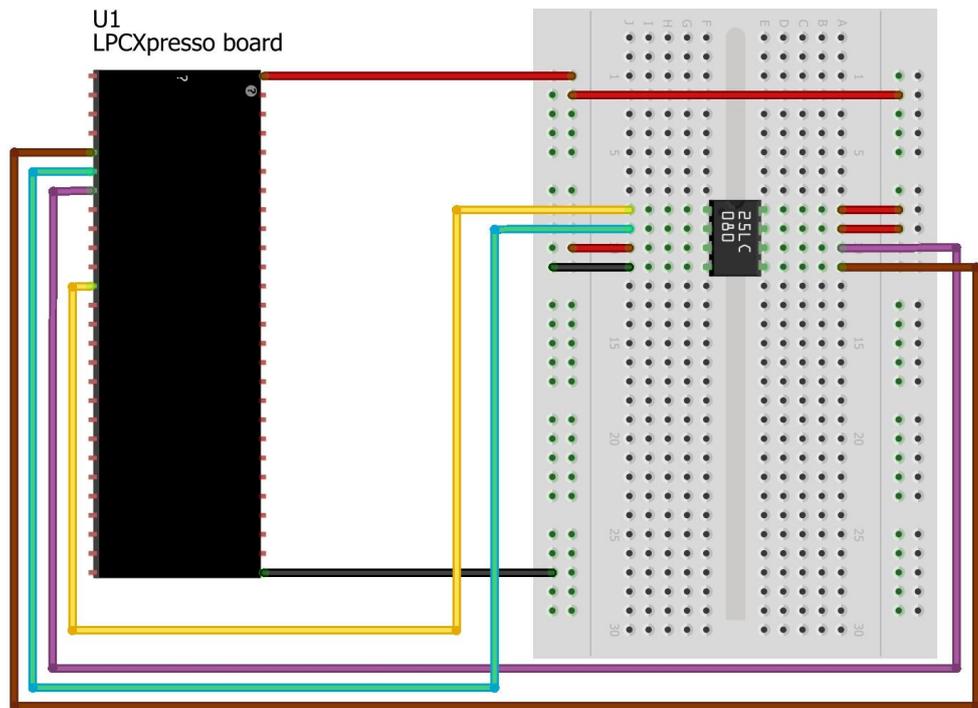
Made with  Fritzing.org

Figure 54 – Breadboard with SPI E2PROM

7.13 Work with Interrupts

In this experiment you will learn how to incorporate interrupts in your program. Interrupts are a powerful concept in embedded programming. It is a way to interrupt the normal program execution flow to service something else quickly. This “something else” is typically a peripheral block that needs to be serviced or it is an external event that needs attention/a reaction.

There is a functional block in the LPC111x that is called the Nested Vectored Interrupt Controller (NVIC). It is an integral part of the Cortex-M0 core. The NVIC can be regarded as a peripheral block, but a special one. It is programmed/setup via registers, just like any other peripheral. It supports 32 interrupt sources and there are four programmable priority levels. Individual interrupts can be masked (i.e., disabled) in the NVIC. It is also possible to generate an interrupt via software - writing in a special register will trigger the specified interrupt.

An indication that the NVIC is a special function block is that all information can be found in chapter 28.6.2 – *Nested Vectored Interrupt Controller* in the LPC111x user's manual. This is the chapter that contains Cortex-M0 core information. Chapter 6 - *LPC111x/LPC11Cx Nested Vectored Interrupt Controller (NVIC)* basically only contains a table (Table 54) that lists the different interrupt sources in the LPC111x. Almost all of the 32 sources are used.

Have a look in file `LPC11xx.h`. It is found in the CMSIS library, in the `inc` sub-directory. Amongst other things, the `typedef` declaration below is found in this file. It lists the names and numbers of the different interrupt sources.

```
...
/* Interrupt Number Definition */
typedef enum IRQn
{
    /***** Cortex-M0 Processor Exceptions Numbers *****/
    NonMaskableInt_IRQn    = -14,    /* 2 Non Maskable Interrupt          */
    HardFault_IRQn         = -13,    /* 3 Cortex-M0 Hard Fault Interrupt  */
    SVCall_IRQn            = -5,     /* 11 Cortex-M0 SV Call Interrupt    */
    PendSV_IRQn            = -2,     /* 14 Cortex-M0 Pend SV Interrupt    */
    SysTick_IRQn           = -1,     /* 15 Cortex-M0 System Tick Interrupt */

    /***** LPC11xx Specific Interrupt Numbers *****/
    WAKEUP0_IRQn           = 0,      /* All I/O pins can be used as wakeup source. */
    WAKEUP1_IRQn           = 1,      /* There are 13 pins in total for LPC11xx      */
    WAKEUP2_IRQn           = 2,
    WAKEUP3_IRQn           = 3,
    WAKEUP4_IRQn           = 4,
    WAKEUP5_IRQn           = 5,
    WAKEUP6_IRQn           = 6,
    WAKEUP7_IRQn           = 7,
    WAKEUP8_IRQn           = 8,
    WAKEUP9_IRQn           = 9,
    WAKEUP10_IRQn          = 10,
    WAKEUP11_IRQn          = 11,
    WAKEUP12_IRQn          = 12,
    SSP1_IRQn              = 14,     /* SSP1 Interrupt                    */
    I2C_IRQn               = 15,     /* I2C Interrupt                     */
    TIMER_16_0_IRQn        = 16,     /* 16-bit Timer0 Interrupt           */
    TIMER_16_1_IRQn        = 17,     /* 16-bit Timer1 Interrupt           */
    TIMER_32_0_IRQn        = 18,     /* 32-bit Timer0 Interrupt           */
    TIMER_32_1_IRQn        = 19,     /* 32-bit Timer1 Interrupt           */
    SSP0_IRQn              = 20,     /* SSP0 Interrupt                    */
    UART_IRQn              = 21,     /* UART Interrupt                    */
    ADC_IRQn               = 24,     /* A/D Converter Interrupt           */
    WDT_IRQn               = 25,     /* Watchdog timer Interrupt          */
    BOD_IRQn               = 26,     /* Brown Out Detect (BOD) Interrupt  */
    EINT3_IRQn             = 28,     /* External Interrupt 3 Interrupt     */
    EINT2_IRQn             = 29,     /* External Interrupt 2 Interrupt     */
    EINT1_IRQn             = 30,     /* External Interrupt 1 Interrupt     */
    EINT0_IRQn             = 31,     /* External Interrupt 0 Interrupt     */
} IRQn_Type;
...
```

An interrupt source is enabled by the call below. The example enables the 16-bit timer #0 interrupt.

```
/* enable 16-bit timer #0 interrupt */
NVIC_EnableIRQ(TIMER_16_0_IRQn);
```

It is also possible to disable an interrupt source.

```
/* disable 16-bit timer #0 interrupt */
NVIC_DisableIRQ(TIMER_16_0_IRQn);
```

Normally it is good system design practice to keep the execution time in the interrupts as short as possible. The actions that are needed immediately are done in the interrupt service routine (ISR). Actions that can wait should be scheduled for later execution in the normal program flow. If it is not possible to keep execution time in an ISR short, nested interrupts can be used. Nested interrupts means that an interrupt can interrupt another interrupt if it has higher priority. Four priority levels (0-3) are supported in the NVIC hardware. The lower the number is, the higher the priority is. Interrupts with the same priority cannot interrupt each other. It is possible to set the priority of an interrupt like this:

```
/* set priority of specified interrupt
   IRQn_Type      , priority (0..3) */
void NVIC_SetPriority(TIMER_16_0_IRQn, (prio<<1)|0x01;
```

Creating an ISR is very simple. The ISR can be written entirely as a C-routine. Have a look in file `cr_startup_lpc11.c`. It is found in the project's `src` sub-directory. Amongst other things this file contains declarations of the ISR:s, as seen below. The functions are called *ISR handlers*, but that is just another name for the same thing – an interrupt service routine, ISR.

```
...
//*****
//
// Forward declaration of the specific IRQ handlers. These are aliased
// to the IntDefaultHandler, which is a 'forever' loop. When the application
// defines a handler (with the same name), this will automatically take
// precedence over these weak definitions
//
//*****
void CAN_IRQHandler      (void) ALIAS(IntDefaultHandler);
void SSP1_IRQHandler    (void) ALIAS(IntDefaultHandler);
void I2C_IRQHandler     (void) ALIAS(IntDefaultHandler);
void TIMER16_0_IRQHandler (void) ALIAS(IntDefaultHandler);
void TIMER16_1_IRQHandler (void) ALIAS(IntDefaultHandler);
void TIMER32_0_IRQHandler (void) ALIAS(IntDefaultHandler);
void TIMER32_1_IRQHandler (void) ALIAS(IntDefaultHandler);
void SSP0_IRQHandler    (void) ALIAS(IntDefaultHandler);
void UART_IRQHandler    (void) ALIAS(IntDefaultHandler);
void ADC_IRQHandler     (void) ALIAS(IntDefaultHandler);
void WDT_IRQHandler     (void) ALIAS(IntDefaultHandler);
void BOD_IRQHandler     (void) ALIAS(IntDefaultHandler);
void PIOINT3_IRQHandler (void) ALIAS(IntDefaultHandler);
void PIOINT2_IRQHandler (void) ALIAS(IntDefaultHandler);
void PIOINT1_IRQHandler (void) ALIAS(IntDefaultHandler);
void PIOINT0_IRQHandler (void) ALIAS(IntDefaultHandler);
void WAKEUP_IRQHandler  (void) ALIAS(IntDefaultHandler);
...

```

If the user program does not contain declarations of these routines/handlers, then they will default to the default interrupt handler (`IntDefaultHandler`). The exact same name of the routines must be used. Below is an example of a custom ISR for 16-bit timer #0.

```
/* My own ISR for 16-bit timer #0 */
void TIMER16_0_IRQHandler (void)
{
    //Service the interrupt and finish with clearing interrupt
    ...
}
```

7.13.1 Lab 12a: Generate IRQ via GPIO

In this experiment an interrupt will be generated from a GPIO input. Rebuild the basic breadboard setup in Figure 13 (on page 39). One LED, controlled by PIO0_2 and one push-button connected to PIO1_5. Let the push-button input generate an interrupt on a falling edge (= pushing the key). Toggle the LED every time the push-button is pressed.

Study chapter 12 – *LPC111x/LPC11Cx General Purpose I/O (GPIO)* in the LPC111x user's manual. Especially note the GPIO features listed (below is an excerpt from user's manual):

- *Each individual port pin can serve as an edge or level-sensitive interrupt request.*
- *Interrupts can be configured on single falling or rising edges and on both edges.*
- *Level-sensitive interrupt pins can be HIGH or LOW-active.*

Register GPIO_nIE (where n is the port number) controls if a pin generates an interrupt, or not. If not, it is said that the interrupt is masked. Default is that all pin interrupts are masked (inactive). For active interrupts (non-masked pins), register GPIO_nIS controls if a pin generates edge or level sensitive interrupts. Register GPIO_nIEV controls if each individual pin interrupt is falling/rising edge active or low/high level active. For edge sensitive interrupts, register GPIO_nIBE controls if a pin is sensitive to one edge (rising or falling) or both.

Whether to select an edge or level sensitive interrupt depends on the application and how the hardware interface works. For the push-button, edge sensitive triggering is suitable since the key press occupation is what should be detected. How long the key is pressed is of no concern (in this experiment). If the interrupt would have been level sensitive the interrupts routine (ISR) would have been activated over and over until the button is no longer pressed. Level sensitive interrupts are suitable when the ISR can reset the interrupt condition (by some action). Note that the ISR must clear the interrupt condition for edge sensitive interrupts (check the GPIO_xIC register).

Study the code below. It is a framework for the experiment. Note that the main loop does nothing - just looping in a forever loop. If power consumption is a concern, it is suitable to place the microcontroller in a low power state. Whenever the interrupt condition occurs the microcontroller will wake up and execute the ISR and then go back to the low power mode.

```

/* Define Interrupt Service Routine for Port #1 */
void PIOINT1_IRQHandler (void)    //name of function is predefined
{
    /* toggle LED on PIO0_2 */
    ...

    /* clear PIO1_5 falling edge interrupt */
    LPC_GPIO1->IC = (1<<5);    //write with bit 5 set to clear interrupt from PIO1_5
}

void main (void)
{
    /* initialize so that PIO1_5 generate an interrupt (falling edge sensitive) */
    ...

    /* enable port #1 interrupt */
    NVIC_EnableIRQ(EINT1_IRQn);

    /* enter forever loop - let interrupt handle processing */
    while(1)
        ;    //here is a potential to go into a low power mode
}

```

Note that due to contact bouncing (inside the pushbutton) sometimes several edges will be detected when the pushbutton is pressed. In this experiment this effect is ignored but to in a real system contact bounce must be handled properly.

7.13.2 Lab 12b: Timer IRQ

In this experiment an interrupt will be generated from a timer. In *Lab 1c: Delay Function – LED Flashing* a simple for-loop was used to create exact a delay function. Recreate the experiment and flash with a LED. Start with a fixed flash pattern; say 5 Hz. Keep the breadboard setup from the previous experiment (see Figure 13, page 39). This experiment is also an extension to *Lab 9a: Create Exact Delay Function*, where a 32-bit timer was user to create exact delay functions.

The code below illustrated a suitable framework to start from.

```

/* Define Interrupt Service Routine for 32-bit timer #1 */
void TIMER32_1_IRQHandler(void) //name of function is predefined
{
    /* toggle LED on PIO0_2 and clear timer interrupt before exiting ISR */
    ...
}

/*****
** Function name:      main
** Descriptions:      The main function
** Parameters:        None
** Returned value:    None
**
**
*****/
void main (void)
{
    /* initialize GPIO as needed */
    ...

    /* setup 32-bit timer #1 to generate continuous interrupts every 200 ms = 5 Hz */
    ...

    /* enable 32-bit timer #1 interrupt */
    NVIC_EnableIRQ(TIMER_32_1_IRQn);

    /* enter forever loop - let interrupt handle processing */
    while(1)
        ;
}

```

Now, expand the functionality of the program and design a program that flash with the LED – 50 ms (milli seconds) on, 150 ms off, 50 ms on and finally and 750 ms off. Continuously repeat this 1000 ms cycle. The suggested program structure is to set the timer interrupt rate high, for example 1000 Hz. That is 1 ms between every interrupt. Check which state the LED should have inside the timer ISR.

```

/* Define Interrupt Service Routine for 32-bit timer #1 */
void TIMER32_1_IRQHandler (void) //name of function is predefined
{
    /* increment millisecond counter */
    msCnt++;

    /* keep counter at one second resolution */
    if (msCnt >= 1000)
        msCnt = 0;

    /* set LED state based on millisecond counter */
    if (...)
    {
        /* set LED */
        ...
    }
    else if (...)
    {
        /* set LED */
        ...
    }
    //etc
    ...
    //clear timer interrupt
}

```

7.13.3 Lab 12c: Timer IRQ with Callback

In this experiment the timer interrupt will call a registered function, called a callback function. It is a commonly used program structure that can be very powerful and flexible.

Create a program that uses a timer callback to control flashing of a LED. Keep the breadboard setup from the previous experiments (see Figure 13, page 39).

Study the code framework below. It outlines how a timer callback functionality can be implemented:

- A function pointer to the callback function is stored.
- A timer is setup to generate an interrupt after a specified time. After triggering the interrupt the timer stops.
- When the timer interrupts occurs, the callback function is called.

```

/* Declare function pointer (to a void-void function) for the callback function */
volatile void (*pCB)(void);

/* Define Interrupt Service Routine for 32-bit timer #1 */
void TIMER32_1_IRQHandler (void) //name of function is predefined
{
    /* check if function pointer is value (not equal to NULL) */
    if(pCB != NULL)
    {
        /* call function pointer = call callback function */
        pCB(); //also valid syntax: (*pCB)();

        /* invalidate function pointer */
        pCB = NULL;
    }

    //stop timer
    ...

    //clear timer interrupt
    ...
}

/*****
** Function name: registerCbAndDelay
** Descriptions: This function setup 32-bit timer #1 to generate
**               an interrupt after specified time and then call a
**               registered callback function.
** parameters: delay in ms and callback function pointer
** Returned value: None
**
*****/
void registerCbAndDelay( uint16_t delayInMS, void (*pF)(void) )
{
    /* register callback function */
    pCB = pF;

    /* setup timer to fire in 'delayInMS' ms */
    ...

    /* enable 32-bit timer #1 interrupt */
    NVIC_EnableIRQ(TIMER_32_1_IRQn);
}

/*****
** Function name: toggleLED
** Descriptions: This function toggles output PIO0_2
** Parameters: None
** Returned value: None
**
*****/
void toggleLED(void)
{
    /* toggle LED on PIO0_2 */
    ...
}

```

```

/*****
** Function name:      main
** Descriptions:     The main function
** Parameters:       None
** Returned value:   None
**
*****/
void main (void)
{
    /* initialize GPIO as needed */
    ...

    /* enter forever loop - let interrupt handle processing */
    while(1)
    {
        /* register callback (to toggle LED) in 200ms */
        registerCbAndDelay(200, &toggleLED);

        /* wait until callback has been called */
        while (pCB != NULL)
            ;
    }
}

```

Place the callback related functions in file `timerCB.c`.

The structure above is not perfect since the caller must wait until the callback has been executed before the next callback can be registered and started. To make it more user friendly, extend the callback timer functionality to also support repeated calls. When registering a new callback one parameter/flag tells if it is a one-time callback or a repeated callback. A new function is then also needed to stop a repeated callback. It would be good programming practice to let the function return an error if there is already an active callback in the system.

A much more flexible and robust framework would allow multiple callbacks to be registered and handled accordingly. Such a framework is however a lot more work and out of the scope for this experiment.

7.13.4 Lab 12d: Nested Interrupts

In this experiment the effect of nested interrupts will be investigated. This experiment is a little combination of Lab 12a and Lab 12b. Use the same breadboard setup as in these experiments, one LED and one push-button.

Setup a repeated timer interrupt to toggle the LED with 5Hz rate. Whenever the push-button is pressed enter a 2 second delay loop (of the old type) in the interrupt service routine. First observe the LED flashing.

Press the push-button. What happens? _____

Yes, the LED will stop toggle for 2 seconds whenever the push-button is pressed. It is exactly what can be expected since the push-button (port #1) ISR will block the timer interrupt. This is an excellent illustration that time spent in an ISR should be kept to a minimum in order not to block other ISR:s from being executed.

Now explicitly set the priority of both ISR:s. Set the priority of the timer ISR higher than for the GPIO ISR. Remember that a lower number (range is 0-3) means higher priority.

Verify that the LED now continues flashing whenever the push-button is pressed.

What is the default priority for all interrupts? _____

There is a function call for reading the priority also. Search in the file `core_cm0.h` after this function.

7.13.5 Lab 12e: Control Dual Digit 7-segment Display

This experiment revisits *Lab 8d: Control Dual Digit 7-segment Display* and *Lab 11b: Control 7-segment Display*. By combining the knowledge from all previous experiments it is now possible to create a system that is quite close to a how this would have been solved in a real system.

Setup a repetitive timer interrupt, say 500 Hz (2 ms between each interrupt). Let the timer ISR update the dual 7-segment display. The ISR alternates which digit that is updated.

The main program just set up the timer ISR and writes the segment outputs in a global variable (that the timer ISR can read when updating the digits).

The suggested program structure for the timer ISR is presented in the code block below.

```
/* Declare variable to store digit outputs */
volatile uint8_t digitSegments[2];

/* Define Interrupt Service Routine for 32-bit timer #1 */
void TIMER32_1_IRQHandler (void) //name of function is predefined
{
    //counter that indicate active digit (numbered 0 and 1)
    static uint8_t activeDigit;

    if (activeDigit == 0)
    {
        //Disconnect anode of digit #0 (pull control signal high)
        ...

        //Send segment outputs (via SPI) for digit 1
        ...

        //Connect anode of digit #1 (pull control signal low)
        ...

        activeDigit = 1;
    }

    else
    {
        //Disconnect anode of digit #1 (pull control signal high)
        ...

        //Send segment outputs (via SPI) for digit 0
        ...

        //Connect anode of digit #0 (pull control signal low)
        ...

        activeDigit = 0;
    }

    //clear interrupt
    ...
}
```

7.14 Work with a Serial Bus – I²C

Note that the breadboard cannot be used in these experiments. The chips used are surface mounted and these must be soldered to the pcb before starting.

In this experiment you will learn how to work with the Inter-Integrated Circuit Bus, or I²C bus for short. It is a multi-master bus for (relatively) low-speed peripherals. The basic clock frequency is 100 kHz but there are newer specifications that support higher speeds, for example 400 kHz that is often supported, called Fast-mode (Fm). Higher frequencies of 1 MHz (Fm+), 3.4 MHz (High-speed mode, Hs) and 5 MHz (Ultra Fast-mode, UFm) also exist but are less widespread.

The I²C bus is a synchronous bus meaning that there is an explicit clock signal. It builds on the master-slave concept where one unit is a master and controls the communication. One slave is addressed on the bus and is the other end of the master-slave communication. There can be many masters on the bus, but only one active at a time.

The I²C bus uses two bidirectional open-drain lines pulled up by resistors:

- SCK: serial clock, the master always generates the clock
- SDA: serial data, the master generates the data when transmitting to the slave. The slave generates the data when transmitting to the master

The picture below illustrates how many masters and slaves can share one I²C bus.

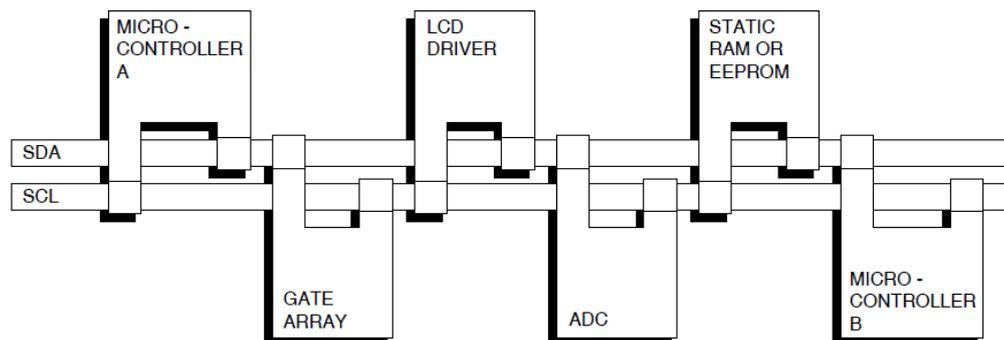


Figure 55 – I²C Bus

For more information about I²C, see <http://en.wikipedia.org/wiki/I%C2%B2C>

There is a lot of details about the I²C bus that have not been covered in this short overview, like how addressing works, how bus arbitration works, how read and write operations work, how acknowledge of data works, etc.

Have a look in chapter 15 - *LPC111x/LPC11Cxx I2C-bus controller* in the LPC111x user's manual for a description of the how the I²C block works. It is more complicated interface than for the timers and SSP peripherals. The basic principle is to send commands to the I²C peripheral block. These commands are carried out in the (external) I²C bus and a status is presented as result. Based on the status the I²C driver gives the next command.

It is not recommended to start from scratch and create an I²C driver. Instead the driver supplied from NXP will be used, see files `i2c.c`/`i2c.h`. Let's investigate the application program interface (API) for this driver. The file `i2c.h` contains (amongst other declarations) the following function declarations:

- **I2CInit()** – this function must be called before the I2C driver is used and any I2C communication can take place. The function initializes the pins (PIO0_4, PIO0_5) to be I2C pins and other necessary initialization. The function has two parameters. The first parameter tells if the I2C interface shall be a master or slave interface. In this case it is a master interface and no further parameter is needed. In case it is a slave interface, the second parameter is the slave address of this interface.

- **I2CRead()** – this function perform a read operation. The function has three parameters. The first is the slave address to communicate with. The second is a buffer pointer to where the read data is copied. The third parameter is the number of bytes to read.
- **I2CWrite()** – this function perform a write operation. The function has three parameters. The first is the slave address to communicate with. The second is a buffer pointer from where to get the data to transfer to the slave. The third parameter is the number of bytes to write/transfer.

7.14.1 Lab 13a: Solder Surface Mounted Components

In this Lab the surface mounted components shall be soldered to the PCB, as a preparation for the following I2C related experiments. Besides the surface mounted components a few connectors are also needed to be soldered for powering and connection to the LPCXpresso LPC111x board. The following components shall be soldered (see chapter 4 for pictures of all different components):

- J1, 2.1mm power jack. This is for allowing an external 5V DC supply to power the board.
- J2, dual 1x27 pos headers for connection to the LPCXpresso LPC111x board.
- J17, mini-B USB connector on bottom side as alternative power source
- Components on schematic page 6
 - Temperature sensor: U6 (LM75), C10 (100nF), R57-R58 (2K)
 - I2C GPIO expander: U7 (PCA9532), C11 (100nF), R43-R56 (2K), LED11-LED18

Figure 56 illustrates where on the PCB the components shall be soldered. Read chapter 6 for information about soldering. In general there are many good tutorials on the Internet on how to solder through-hole components as well as surface mounted components. Just search with your favorite search engine.

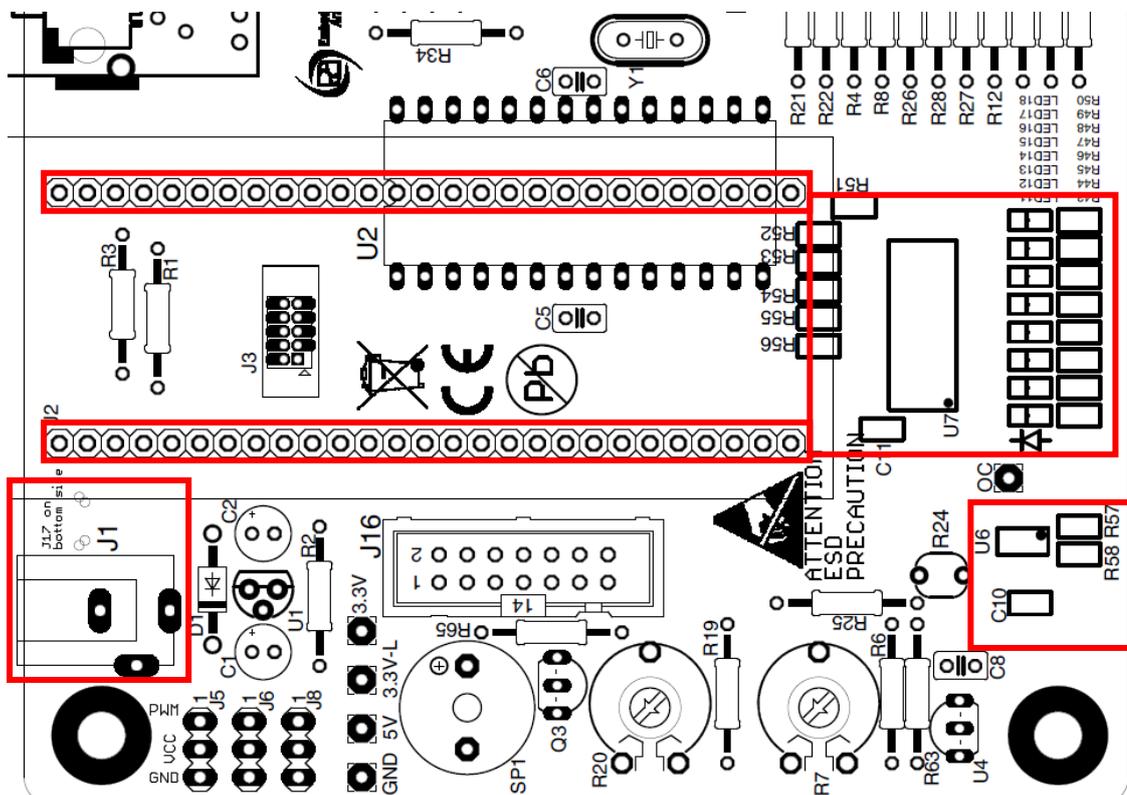


Figure 56 – Surface Mounted Components on the LPCXpresso Experiment Kit PCB

7.14.2 Lab 13b: Read LM75 Temperature Sensor

In this experiment a temperature sensor, LM75, shall be sampled and the temperature presented. It is essential to study the LM75 datasheet before writing any code. The LM75 has a simple interface and luckily no register initialization is needed before it is possible to read the temperature. It is just a matter of reading from the correct register.

The code below presents two functions. One for reading the temperature and one for writing in configuration registers. Complete the last statement in function `lm75a_readTemp()` to calculate the correct temperature. Create a semihosting application that samples the temperature every third second and prints the result on the console.

```
#include "i2c.h"

#define LM75B_I2C_ADDR 0x90
#define LM75B_REG_TEMP 0x00
#define LM75B_REG_CMD 0x01

/*****
 *
 * Description:
 *   Read temperature register of LM75B
 *
 * Params: None
 * Returns: Temperature * 100 in integer format
 *
 *****/
int32_t lm75b_readTemp(void)
{
    uint8_t cmd, temp[2];
    int32_t t = 0;

    cmd = LM75B_REG_TEMP;
    I2CWrite(LM75B_I2C_ADDR, &cmd, 1);
    I2CRead(LM75B_I2C_ADDR, &temp[0], 2);

    /* 11 MSB bits used. Celsius is calculated as Temp data * 1/8 */
    t = ((temp[0] << 8) | (temp[1]));

    /* Return temperature times 100, e.g., in 0.01 degrees */
    return ...;
}

/*****
 *
 * Description:
 *   Write to config register of LM75B
 *
 * Params: Config byte
 * Returns: None
 *
 *****/
void lm75b_config(int8_t config)
{
    uint8_t cmd[2];

    cmd[0] = LM75B_REG_CMD;
    cmd[1] = config;

    I2CWrite(LM75B_I2C_ADDR, &cmd[0], 2);
}

/*****
** Function name:    main
** Descriptions:    The main function
** Parameters:      None
** Returned value:  None
**
*****/
```

```

*****/
void main (void)
{
  /* initialize I2C as needed */
  I2CInit( I2CMaster, 0 );

  /* enter forever loop */
  while(1)
  {
    /* read temperature and print result */
    ...

    /* wait 3 seconds */
    ...
  }
}

```

Place the LM75 related code in file `lm75.c`.

7.14.3 Lab 13c: Control LEDs via PCA9532

In this experiment a GPIO expansion chip, PCA9532, shall be used. The chip can also generate PWM waveforms to for example dim LEDs. It is essential to study the PCA9532 datasheet before writing any code. The PCA9532 chip has a more complex interface than the LM75. More registers must be controlled. There are 16 I/O pins and 10 registers in the chip:

- Two registers are used for reading the 16 inputs (two bytes).
- There are two PWM generators in the chip. Two registers are needed to control each generator, so four registers in total for this.
- Four registers are used to control the 16 pins if they are outputs. 2 bits per pin, which means that one byte can control 4 pins – resulting in four registers to control 16 pins. A pin can have one of the following four states:
 - Actively driven low.
 - High impedance where the pin is typically driven high by an external pullup resistor. The pin can also be an input in this state.
 - Driven by PWM generator #0, alternating between actively driven low and high-impedance.
 - Driven by PWM generator #1.

The external LEDs are connected via the cathode to the PCA9532 chip. This is because the chip can only sink current.

Below is a code framework for controlling the 16 outputs via function `pca9532_setLeds(...)`.

```

#define PCA9532_I2C_ADDR    0xC0

#define PCA9532_INPUT0 0x00
#define PCA9532_INPUT1 0x01
#define PCA9532_PSC0   0x02
#define PCA9532_PWM0   0x03
#define PCA9532_PSC1   0x04
#define PCA9532_PWM1   0x05
#define PCA9532_LS0    0x06
#define PCA9532_LS1    0x07
#define PCA9532_LS2    0x08
#define PCA9532_LS3    0x09

#define PCA9532_AUTO_INC 0x10

/*****
 * Defines and typedefs
 *****/

```

```

#define LS_MODE_ON      0x01
#define LS_MODE_BLINK0 0x02
#define LS_MODE_BLINK1 0x03

/*****
 * Local variables
 *****/
static uint16_t blink0Shadow = 0;
static uint16_t blink1Shadow = 0;
static uint16_t ledStateShadow = 0;

/*****
 * Local Functions
 *****/
static void setLsStates(uint16_t states, uint8_t* ls, uint8_t mode)
{
#define IS_LED_SET(bit, x) ( ((x) & (bit)) != 0 ) ? 1 : 0 )

    int i = 0;

    for (i = 0; i < 4; i++) {
        ls[i] |= ( (IS_LED_SET(0x0001, states)*mode << 0)
                | (IS_LED_SET(0x0002, states)*mode << 2)
                | (IS_LED_SET(0x0004, states)*mode << 4)
                | (IS_LED_SET(0x0008, states)*mode << 6) );
        states >>= 4;
    }
}

static void setLeds(void)
{
    uint8_t buf[5];
    uint8_t ls[4] = {0,0,0,0};
    uint16_t states = ledStateShadow;

    /* LEDs in On/Off state */
    setLsStates(states, ls, LS_MODE_ON);

    /* set the LEDs that should blink */
    setLsStates(blink0Shadow, ls, LS_MODE_BLINK0);
    setLsStates(blink1Shadow, ls, LS_MODE_BLINK1);

    buf[0] = PCA9532_LS0 | PCA9532_AUTO_INC;
    buf[1] = ls[0];
    buf[2] = ls[1];
    buf[3] = ls[2];
    buf[4] = ls[3];
    I2CWrite(PCA9532_I2C_ADDR, buf, 5);
}

/*****
 * Public Functions
 *****/

/*****
 *
 * Description:
 *   Set LED states (on or off).
 *
 * Params:
 *   [in] ledOnMask - The LEDs that should be turned on. This mask has
 *                   priority over ledOffMask
 *   [in] ledOffMask - The LEDs that should be turned off.
 *****/
void pca9532_setLeds (uint16_t ledOnMask, uint16_t ledOffMask)
{
    /* turn off leds */
    ledStateShadow &= (~(ledOffMask) & 0xffff);

    /* ledOnMask has priority over ledOffMask */
    ledStateShadow |= ledOnMask;

    /* turn off blinking */
    blink0Shadow &= (~(ledOffMask) & 0xffff);
}

```

```
    blink1Shadow &= (~(ledOffMask) & 0xffff);  
    setLeds();  
}
```

Add functionality to control the PWM generators and functions to direct the PWM signals to specific pins. Place the PCA9532 related code in file `pca9532.c`.

Create an application that performs a running one pattern on the eight connect LEDs.

Also create a program that can demonstrate dimming on the LEDs with the help of the PWM generators on the PCA9532 chip.

7.15 Work with a Serial Bus – UART

In this experiment you will learn how to work with the Universal Asynchronous Receiver/Transmitter, or UART for short. The term *asynchronous* refers to the fact that no explicit clock signal is transmitted. The transmitter and receiver must agree beforehand on the bit rate, i.e., how long time a transmitted bit shall take. The idle state (no transmission) is a high signal. Transmission begins with a start bit, which is low. The negative edge is detected by the receiver and 1.5 bit periods after this, bit sampling begins. Eight data bits are sampled. The least significant bit (LSB) is typically transmitted first. An optional parity bit is then transmitted (for error checking of the data bits). Often this bit is omitted if the transmission channel is assumed to be noise free or if there are error checking higher up in the protocol layers. The transmission is ended by a stop bit. Typically one bit, but 1.5 and 2 bits are sometimes also used. Most common for inter-board communication is 8N1, meaning 8 data bits, no parity and one stop bit.

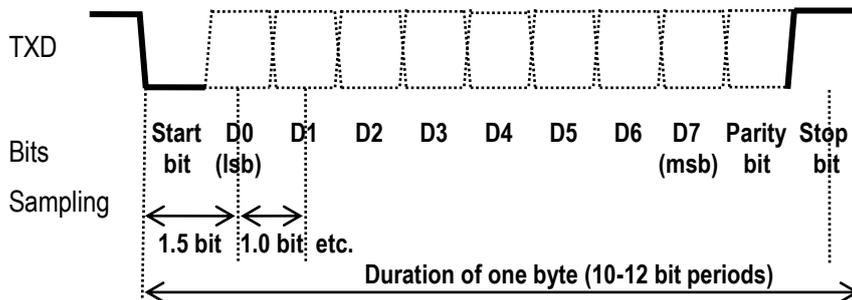


Figure 57 – UART Communication

On a side note, there are methods to determine the bit rate of a received signal but that is out of scope for this experiment.

An UART channel consists of two signals (besides ground):

- TXD: transmit data, direction from transmitter to receiver. This is an output.
- RXD: receive data, direction from transmitter to receiver. This is an input.

TXD and RXD are crossed between transmitter and receiver, i.e., TXD is connected to RXD and vice versa. For more information about asynchronous serial communication, see http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter and http://en.wikipedia.org/wiki/Asynchronous_serial_communication.

Note and understand the difference between the signaling method (asynchronous serial communication) and standards of voltage signaling. The signal drawn in Figure 57 illustrates the signal to/from the UART peripheral inside the LPC111x. It is a 3.3V logic signal. This is common for communication between units on the same board, or closely mounted boards. RS232 is a common signaling standard with large voltage swings (+- 3-15V) that is used between units that are physically apart. RS422 and RS485 are other commonly used signaling standards.

Communication is normally point-to-point, meaning that a transmitter sends data to one receiver. There are signaling standards that also supports network topologies (for example RS422 and RS485). Higher protocol layers must then be involved in implementing addressing schemes between the nodes.

Note that this experiment requires a UART-to-USB cable from FTDI (TTL-232R-3V3, Digikey: 768-1015-ND or Mouser: 895-TTL-232R-3V3). This cable is a bridge between a UART channel and USB communication. Via USB it creates a virtual COM port on a PC. The UART communication is tunneled over USB to the PC. When plugging in the USB connector on a PC a driver will be installed. See FTDI's installation guides for details how to install the driver for different operating systems: <http://www.ftdichip.com/Support/Documents/InstallGuides.htm>

The UART signals from the LPC111x are made available on connector J18, see schematic below. Signal GPIO_5-TXD carries the transmitted UART signal and GPIO_6-RXD is the received UART signal. The experiments can take place on the pcb or on a breadboard. Resistor R62 has been added for protection in case GPIO_6-RXD is programmed (by mistake) as an output. If that would happen, R62 limits the currents to safe levels so no output gets damaged.

FTDI UART-to-USB Connector

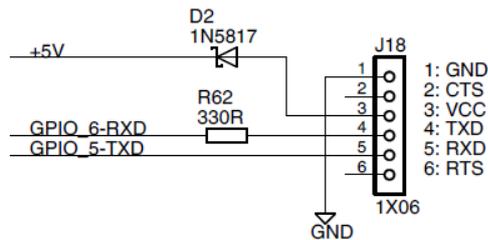


Figure 58 – J18, D2 and R62 on Schematic Page 7

Note orientation of the 6 pos connector of the cable. The black cable is positioned on pin 1 and is ground. Figure 59 illustrates correct orientation when mounting the cable on the pcb.

Also note that the cable can actually power the system since the FTDI cable can supply a +5V voltage. Diode D2 is included in case supply comes from multiple sources (for example, 2.1mm power jack J1 or USB connector J17).

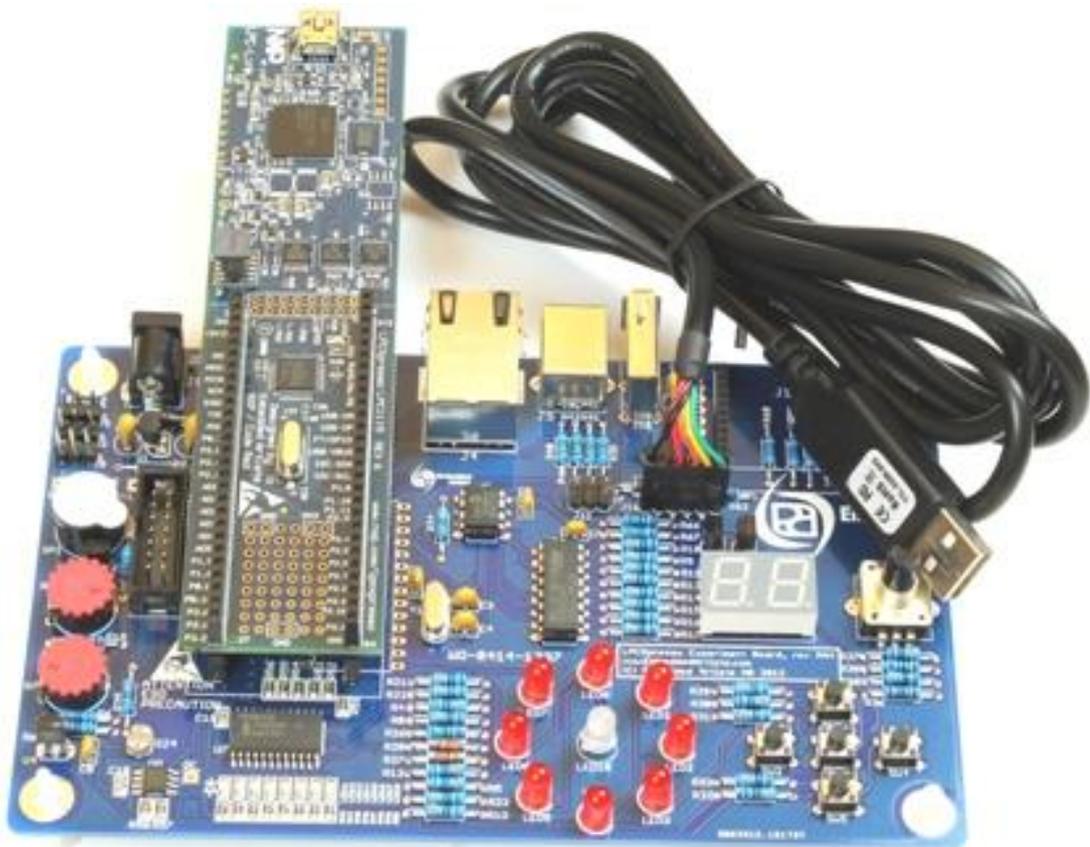


Figure 59 – LPCXpresso Experiment Kit PCB with UART-to-USB Cable

On the PC side, a terminal application is needed. A terminal application connects to a COM port and displays everything received and also allows sending data from the application (via keyboard and sending a file). There are a few good terminal applications:

- TeraTerm (which is recommended), <http://sourceforge.jp/projects/ttssh2/files>
- PuTTY, <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
- Terminal by Bray, <http://sites.google.com/site/braypp/terminal>
- RealTerm, <http://sourceforge.net/projects/realterm/files/>

Download and install the selected terminal application. The next step is to configure the application. Typical configuration settings are selecting COM port, setting bit rate (for example 9600 bps), set if parity is used and number of stop bits. Flow control is another setting that is common. Select *None* for this setting. Other settings require either additional hardware or software support.

For TeraTerm, select *New Connection* in the *File* menu. Select the COM port that appears when the FTDI UART-to-USB cable is connected to the PC. Click OK. The screenshot below illustrates the dialog window for setting up a new serial connection.

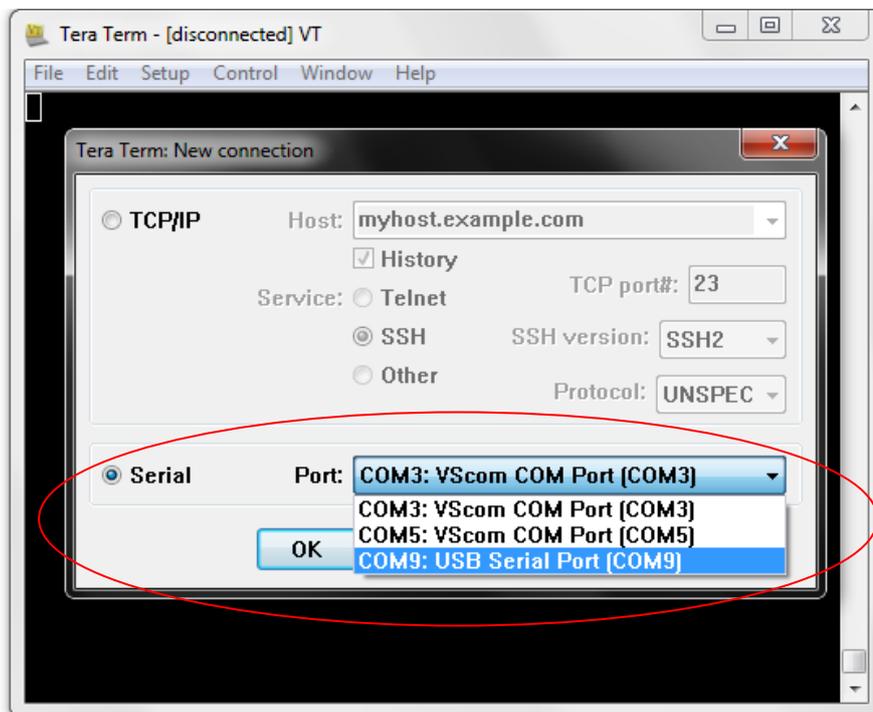


Figure 60 – TeraTerm Configuration Step 1

To set the bit rate and other relevant settings for the serial channel, go to *Setup* menu and select *Serial Port*. A dialog, like illustrated in the screenshot below, opens. Make sure *Flow control* is set to none.

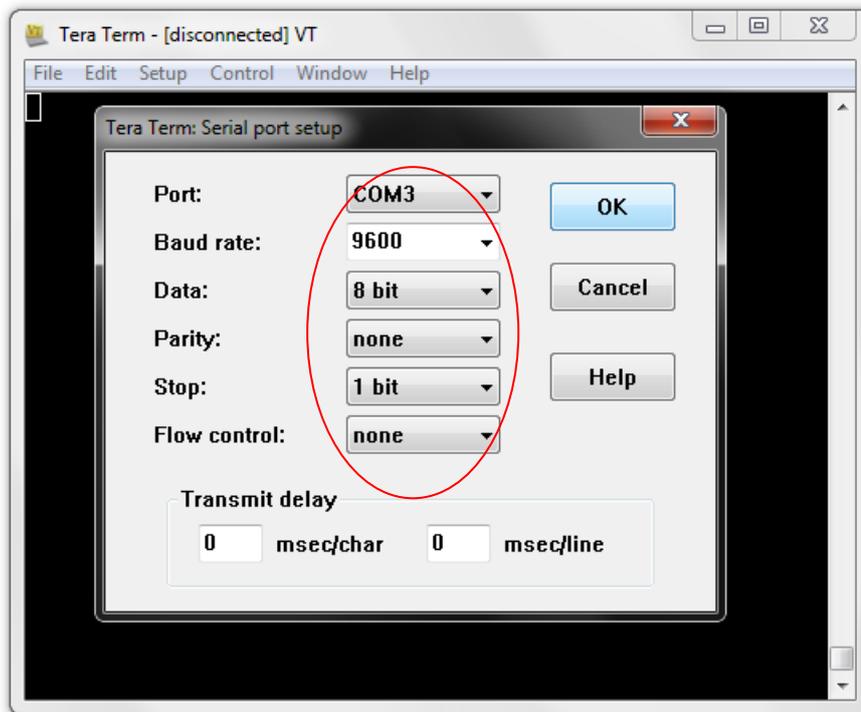


Figure 61 – TeraTerm Configuration Step 2

There are many different settings for how the terminal program shall behave (i.e., interpret received characters). Some adjustments might be needed, for example when to start displaying received characters on a new line. Under menu *Setup*, sub-menu *Terminal setup* it is possible to control these things. The screenshot below illustrates the settings possible for when a new-line shall be performed on received characters. A common setting is *LF*, but it also depends on which character the LPC111x application outputs.

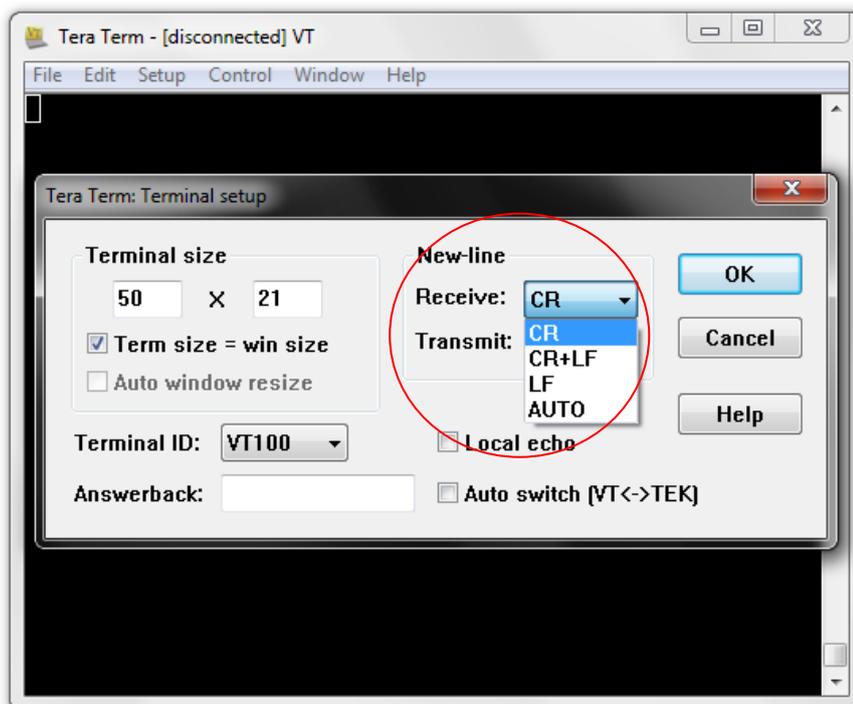


Figure 62 – TeraTerm Configuration Step 3

Have a look in chapter 13 - *LPC111x/LPC11Cx UART* in the LPC111x user's manual for a description of the how the UART block works. The basic principles are the same as for the SPI block – it is a serial shift register for transmitting and receiving. The difference is that for the UART block it is more complex with for example separate shift registers for transmitting and receiving and more flexibility in setting the bit rates. Some features like auto-flow control, auto-baud, modem signaling and RS-485 functionality will not be covered by these experiments.

The transmit signal (TXD) is available on pin PIO1_7 (signal GPIO_5-TXD in the schematic) and the receive signal (RXD) is available on pin PIO1_6 (signal GPIO_6-RXD in the schematic).

Below are some functions to get the UART functionality started. The `UARTInit()` function will initialize the pin-muxing, enable the UART peripheral, setup the bit rate and empty the FIFO:s.

```
#include "LPC11xx.h"
#include "uart.h"

#define LSR_RDR    0x01
#define LSR_OE    0x02
#define LSR_PE    0x04
#define LSR_FE    0x08
#define LSR_BI    0x10
#define LSR_THRE  0x20
#define LSR_TEMT  0x40
#define LSR_RXFE  0x80

/*****
** Function name:  UARTInit
**
** Descriptions:  Initialize UART0 port, setup pin select,
**                clock, parity, stop bits, FIFO, etc.
**
** parameters:    UART baudrate
** Returned value: None
**
*****/
void UARTInit(uint32_t baudrate)
{
    uint32_t Fdiv;
    uint32_t regVal;

    LPC_IOCON->PIO1_6 &= ~0x07;      /* UART I/O config */
    LPC_IOCON->PIO1_6 |= 0x01;      /* UART RXD */
    LPC_IOCON->PIO1_7 &= ~0x07;
    LPC_IOCON->PIO1_7 |= 0x01;      /* UART TXD */

    /* Enable UART clock */
    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<12);
    LPC_SYSCON->UARTCLKDIV = 0x1;   /* divided by 1 */

    LPC_UART->LCR = 0x83;           /* 8 bits, no Parity, 1 Stop bit */
    regVal = LPC_SYSCON->UARTCLKDIV;
    Fdiv = (((SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV)/regVal)/16)/baudrate;

    LPC_UART->DLM = Fdiv / 256;
    LPC_UART->DLL = Fdiv % 256;
    LPC_UART->LCR = 0x03;           /* DLAB = 0 */
    LPC_UART->FCR = 0x07;           /* Enable and reset TX and RX FIFO. */

    /* Read to clear the line status. */
    regVal = LPC_UART->LSR;

    /* Ensure a clean start, no data in either TX or RX FIFO. */
    while ( (LPC_UART->LSR & (LSR_THRE|LSR_TEMT)) != (LSR_THRE|LSR_TEMT) );
    while ( LPC_UART->LSR & LSR_RDR )
    {
        regVal = LPC_UART->RBR;     /* Dump data from RX FIFO */
    }

    return;
}

/*****
```

```

** Function name:  UARTSendChar
**
** Descriptions:  Send a byte/char of data to the UART 0 port
**
** parameters:    byte to send
** Returned value: None
**
*****/
void UARTSendChar(uint8_t toSend)
{
    /* THREE status, contain valid data */
    while ( !(LPC_UART->LSR & LSR_THRE) )
        ;
    LPC_UART->THR = toSend;
}

/*****/
** Function name:  UARTReceive
**
** Descriptions:  Receive a block of data from the UART 0 port based
**                on the data length
**
** parameters:    buffer pointer, data length
** Returned value: Number of received bytes
**
*****/
uint32_t UARTReceive(uint8_t *buffer, uint32_t length, uint32_t blocking)
{
    uint32_t recvd = 0;
    uint32_t toRecv = length;

    if (blocking) {
        while (toRecv) {
            /* wait for data */
            while ( !(LPC_UART->LSR & LSR_RDR) );

            *buffer++ = LPC_UART->RBR;
            recvd++;
            toRecv--;
        }
    }
    else {
        while (toRecv) {
            /* break if no data */
            if ( !(LPC_UART->LSR & LSR_RDR) ) {
                break;
            }

            *buffer++ = LPC_UART->RBR;
            recvd++;
            toRecv--;
        }
    }
    return recvd;
}

```

Function **UARTSendChar ()** transmits one byte/char of data. **UARTReceive ()** is a function that can receive data either blocking or non-blocking. Blocking means that the processor spends all time idle waiting (in the function call) for the wanted number of characters to be received. Non-blocking means that the function returns with the number of available characters that was/were received. It will be somewhere between zero (0) and **length** number of characters. Another name for non-blocking is *asynchronous function call*. Blocking calls can also be called *synchronous function calls*.

Place the UART related code in file **uart.c**, and place the function prototype declarations in file **uart.h**.

7.15.1 Lab 14a: Transmitting and Receiving via the UART

Expand the `UARTSendChar()` function to `UARTSendString(uint8_t *pStr)` function (transmits a zero-terminated string – the terminating zero is not transmitted) and `UARTSendBuffer(uint8_t *pBuf, uint16_t length)` functions.

Create a small program that makes use of these transmission functions. Also let the program echo every received character.

Connect the FTDI cable and verify that the program works as intended.

Test to echo a longer string: "Received: x" (where x is the received char) from the LPC111x. Type characters on the PC terminal program and observe the echoed response from the LPC111x.

Now test to send a series of 100 characters back to back from the PC. This is for example done by sending a 100 byte long file. Select *Send file* in the *File* menu. What will happen?

Every received character results in several characters echoed back to the PC. These echoed characters will take longer time than the time of the (originally) received character. Characters are received at full speed (back-to-back) and soon both transmit and received FIFOs will be full. Received characters will start to be missed.

The solution is flow control. A receiver must be able to inform a transmitter that it must wait for a while before transmitting more characters. One commonly used software solution for this is called Xon/Xoff flow control, see: http://en.wikipedia.org/wiki/Software_flow_control for more information. A commonly used hardware solution is called RTS/CTS flow control, see: http://en.wikipedia.org/wiki/Flow_control_%28data%29#Hardware_flow_control

7.15.2 Lab 14b: Direct printf() to UART

In exercise Lab 4a-4d, semihosting was explored. In this experiment the `printf()` output will be sent to the UART communication channel. Remember that the C runtime library had to be of the correct type for semihosting to work. Have a look at Figure 21 and make sure the project settings select *Redlib (nohost)* as the C runtime library for this exercise. There are hooks in Redlib for directing the `printf()`-output to any wanted communication channel – the UART in this case.

The two simple functions below is all that is needed to direct the `printf()`-output to the UART and also to let `scanf()`-input come from the UART.

```
#include "stdio.h"
#include "uart.h"
...

//use UART for printf
int __sys_write(int iFileHandle, char *pcBuffer, int iLength)
{
    UARTSendBuffer((uint8_t *)pcBuffer,iLength);    //send data buffer to UART
    return iLength;
}

int __sys_readc(void)
{
    char c;
    UARTReceive((uint8_t*)&c, 1, TRUE);
    return (int)c;
}
```

Place the two functions above in a file called `retarget.c`.

Create a program that outputs a message, with the help of `printf()`, on the UART channel and receive the message on a terminal program on a PC. Also let the program verify that `scanf()` works.

Remember that the UART must still be initialized before `printf()/scanf()` are used.

7.15.3 Lab 14c: Interrupt driven UART handling and ring buffers

Blocking function calls can be problematic since it can block other activities in a system. A way to handle this is to create circular buffers, both for received characters and transmission. An interrupt handler place received characters in the receive buffer. The program can then peek into the circular buffer to check if there are any received characters. If not, execution can continue with other tasks.

Below is code the implements UART receive functionality with the help of interrupts and circular buffers. The interrupt routine (ISR) handle both receive and transmit interrupts. Study the code to understand how it works.

```
#include "lpc11xx.h"
#include "uart.h"

//size of transmit buffer - size MUST be power of two
#define TX_BUFFER_SIZE 256
#define TX_BUFFER_MASK (TX_BUFFER_SIZE-1)

//size of receive buffer - size MUST be power of two
#define RX_BUFFER_SIZE 256
#define RX_BUFFER_MASK (RX_BUFFER_SIZE-1)

#define LSR_RDR    0x01
#define LSR_OE    0x02
#define LSR_PE    0x04
#define LSR_FE    0x08
#define LSR_BI    0x10
#define LSR_THRE  0x20
#define LSR_TEMT  0x40
#define LSR_RXFE  0x80

#define IER_RBR    0x01
#define IER_THRE  0x02
#define IER_RLS    0x04

#define IIR_PEND  0x01
#define IIR_RLS   0x03
#define IIR_RDA   0x02
#define IIR_CTI   0x06
#define IIR_THRE  0x01

static volatile uint8_t  txBuf[TX_BUFFER_SIZE];
static volatile uint32_t txHead = 0;
static volatile uint32_t txTail = 0;
static volatile uint8_t  txRunning = FALSE;

static volatile uint8_t  rxBuf[RX_BUFFER_SIZE];
static volatile uint32_t rxHead = 0;
static volatile uint32_t rxTail = 0;

/*****
** Function name:  UARTInit
**
** Descriptions:  Initialize UART0 port, setup pin select,
**                clock, parity, stop bits, FIFO, etc.
**
** parameters:    UART baudrate
** Returned value: None
**
*****/
void UARTInit(uint32_t baudrate)
```

```

{
uint32_t Fdiv;
uint32_t regVal;

NVIC_DisableIRQ(UART_IRQn);

LPC_IOCON->PIO1_6 &= ~0x07;          /* UART I/O config */
LPC_IOCON->PIO1_6 |= 0x01;          /* UART RXD */
LPC_IOCON->PIO1_7 &= ~0x07;
LPC_IOCON->PIO1_7 |= 0x01;          /* UART TXD */

/* Enable UART clock */
LPC_SYSCON->SYSAHBCLKCTRL |= (1<<12);
LPC_SYSCON->UARTCLKDIV = 0x1;      /* divided by 1 */

LPC_UART->LCR = 0x83;                /* 8 bits, no Parity, 1 Stop bit */
regVal = LPC_SYSCON->UARTCLKDIV;
Fdiv = (((SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV)/regVal)/16)/baudrate;

LPC_UART->DLM = Fdiv / 256;
LPC_UART->DLL = Fdiv % 256;
LPC_UART->LCR = 0x03;               /* DLAB = 0 */
LPC_UART->FCR = 0x07;               /* Enable and reset TX and RX FIFO. */

/* Read to clear the line status. */
regVal = LPC_UART->LSR;

/* Ensure a clean start, no data in either TX or RX FIFO. */
while ( (LPC_UART->LSR & (LSR_THRE|LSR_TEMT)) != (LSR_THRE|LSR_TEMT) );
while ( LPC_UART->LSR & LSR_RDR )
{
    regVal = LPC_UART->RBR; /* Dump data from RX FIFO */
}

//initialize the transmit data queue
txHead    = 0;
txTail    = 0;
txRunning = FALSE;

//initialize the receive data queue
rxHead    = 0;
rxTail    = 0;

/* Enable the UART Interrupt */
NVIC_EnableIRQ(UART_IRQn);

LPC_UART->IER = IER_RBR | IER_THRE | IER_RLS; /* Enable UART interrupt */
}

/*****
** Function name:   UART_IRQHandler
**
** Descriptions:   UART interrupt handler
**
** parameters:     None
** Returned value:  None
**
*****/
void UART_IRQHandler(void)
{
    volatile uint8_t IIRValue, LSRValue, statusReg;
    uint8_t Dummy = Dummy;
    volatile uint32_t tmpHead;
    volatile uint32_t tmpTail;

    statusReg = IIRValue = LPC_UART->IIR;
    IIRValue >>= 1;          /* skip pending bit in IIR */
    IIRValue &= 0x07;        /* check bit 1~3, interrupt identification */
    if (IIRValue == IIR_RLS) /* Receive Line Status */
    {
        LSRValue = LPC_UART->LSR;
        /* Receive Line Status */
        if (LSRValue & (LSR_OE | LSR_PE | LSR_FE | LSR_RXFE | LSR_BI))
        {
            /* There are errors or break interrupt */

```

```

    /* Read LSR will clear the interrupt */
    Dummy = LPC_UART->RBR; //Dummy read on RX to clear interrupt, then bail out
    return;
}
if (LSRValue & LSR_RDR) /* Receive Data Ready */
{
    /* If no error on RLS, normal ready, save into the data buffer. */
    /* Note: read RBR will clear the interrupt */
    tmpHead = (rxHead + 1) & RX_BUFFER_MASK;
    rxHead = tmpHead;

    if(tmpHead == rxTail)
        tmpHead = LPC_UART->RBR; //dummy read to reset IRQ flag
    else
        rxBuf[tmpHead] = LPC_UART->RBR; //will reset IRQ flag
}
}
else if (IIRValue == IIR_RDA) /* Receive Data Available */
{
    /* Receive Data Available */
    tmpHead = (rxHead + 1) & RX_BUFFER_MASK;
    rxHead = tmpHead;

    if(tmpHead == rxTail)
        tmpHead = LPC_UART->RBR; //dummy read to reset IRQ flag
    else
        rxBuf[tmpHead] = LPC_UART->RBR; //will reset IRQ flag
}
else if (IIRValue == IIR_CTI) /* Character timeout indicator */
{
    /* Character Time-out indicator */
    ; //functionality not implemented
}
else if (IIRValue == IIR_THRE) /* THRE, transmit holding register empty */
{
    //check if all data is transmitted
    if (txHead != txTail)
    {
        uint32_t bytesToSend;

        if (statusReg & 0xc0)
            bytesToSend = 16; //FIFO enabled
        else
            bytesToSend = 1; //no FIFO enabled

        do
        {
            //calculate buffer index
            tmpTail = (txTail + 1) & TX_BUFFER_MASK;

            txTail = tmpTail;
            LPC_UART->THR = txBuf[tmpTail];
        } while((txHead != txTail) && --bytesToSend);
    }

    //all data has been transmitted
    else
    {
        txRunning = FALSE;
        LPC_UART->IER &= ~IER_THRE; //disable TX IRQ
    }
}
}

/*****
** Function name: UARTSendChar
**
** Descriptions: Send a byte/char of data to the UART 0 port
**
** parameters: byte to send
** Returned value: None
**
*****/
void UARTSendChar(uint8_t toSend)
{

```

```

uint32_t tmpHead;

//calculate head index
tmpHead = (txHead + 1) & TX_BUFFER_MASK;

//wait for free space in buffer
while(tmpHead == txTail)
    ;

//disable TX IRQ
LPC_UART->IER &= ~IER_THRE;

if(txRunning == TRUE)
{
    txBuf[tmpHead] = toSend;
    txHead         = tmpHead;
}
else
{
    txRunning = TRUE;

    /* Extra check - should not be needed: THRE status, contain valid data */
    while ( !(LPC_UART->LSR & LSR_THRE) )
        ;

    LPC_UART->THR = toSend;
}

//enable TX IRQ
LPC_UART->IER |= IER_THRE;
}

/*****
** Function name:  UARTGetCharBlock
**
** Descriptions:  Receive a char from UART 0
**
** parameters:    None
** Returned value: Received char
**
*****/
uint8_t UARTGetCharBlock(void)
{
    //exercise to implement this function...
}

/*****
** Function name:  UARTGetChar
**
** Descriptions:  Receive a char from UART 0
**
** parameters:    pointer to where to store received char
** Returned value: TRUE if char received, else FALSE
**
*****/
uint8_t UARTGetChar(uint8_t *pRxChar)
{
    uint32_t tmpTail;

    /* check if buffer is empty */
    if(rxHead == rxTail)
        return FALSE;

    tmpTail = (rxTail + 1) & RX_BUFFER_MASK;
    rxTail = tmpTail;

    *pRxChar = rxBuf[tmpTail];
    return TRUE;
}

```

Place the UART related code in file `uart.c`, and place the function prototype declarations in file `uart.h`. Do not forget to remove the `retarget.c` file and change back C runtime library to *Redlib (semihost)*.

The code is quite complex and builds on two circular buffers. The receive and transmit buffers can have different sizes, but they must be a power of two. The reason for this is the special mask operations (bitwise-AND with size minus 1). Characters to be transmitted should be placed in a circular buffer. If transmission is not active, start transmission again with the first character in the buffer. As soon as a character has been transmitted the interrupt handler checks if there are more characters to be transmitted in the buffer. If no more, disable the transmission interrupt.

Extend the code above to also implement a `uint8_t UARTGetCharBlock(void)` function that blocks until a char has been received and returns the received character. Let the new function make use of the `UARTGetChar()` function.

Also update the `UARTSendString()` and `UARTSendBuffer()` functions from Lab14a. The suggested function prototypes are as below. These functions are needed on future experiments (with RF modules).

```

/*****
** Function name: UARTSendString
**
** Descriptions: Send a null-terminated string to UART 0 port
**
** parameters: byte to send and if call should wait for transfer to complete
** Returned value: None
**
*****/
void UARTSendString(uint8_t *pStr, uint8_t blocking);

/*****
** Function name: UARTSendBuffer
**
** Descriptions: Send a number of bytes/chars of data to UART 0 port
**
** parameters: data to send, number of bytes and if call is blocking
** Returned value: None
**
*****/
void UARTSendBuffer(uint8_t *pBuf, uint16_t length, uint8_t blocking);

```

Create an application that demonstrates receive and transmit circular buffers. The receive overflow problem in the previous experiment can for example be handled to some extent if the buffers are large enough.

7.16 Extra: Work with RF-module

In this experiment you will learn how to work with radio modules.

Note that the breadboard cannot be used in these experiments. The RF module connectors have 2.0 mm pitch as opposed to the 2.54 mm pitch found on breadboards. Also note that the radio modules used in these experiments are not included and must be purchased separately.

The RF module interface is found on schematic page 7, also replicated in the picture below. All components in Figure 63 must be soldered. J2, the LPCXpresso board connector on schematic page 2, must also be soldered.

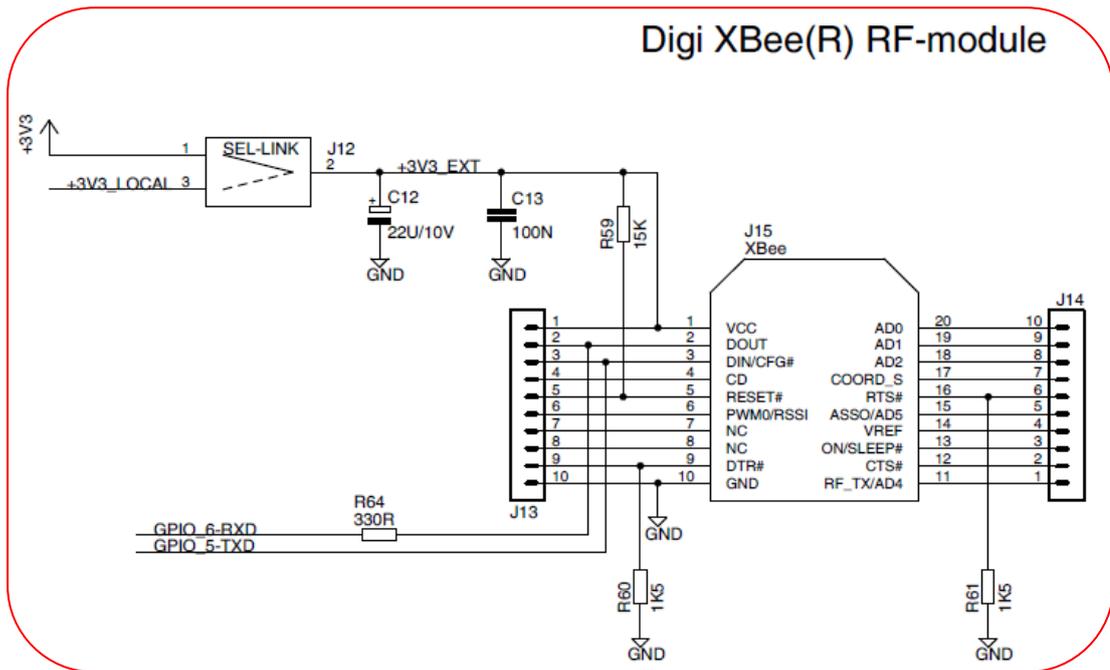


Figure 63 – RF Module Interface on Schematic Page 7

Many RF modules come in the XBee® physical form factor. An UART interface is connected for communication with the RF module. All 20 pins are however available via connectors J13/J14 in case some other pins must be connected for more advanced experiments where more functionality in the modules is utilized. Figure 64 illustrates how an XBee module is mounted in J15. Note that J12 should normally have a shorting jumper in position 1-2, the right position as illustrated in the picture below.

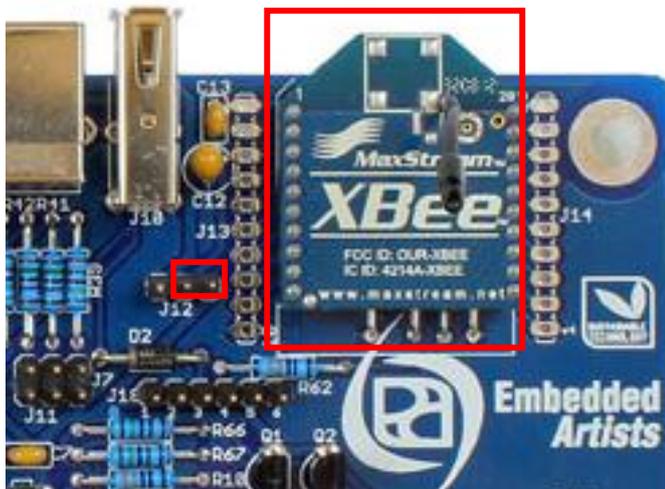


Figure 64 – XBee® Module Mounted in J15

A shorting jumper in position 1-2 of J12 means that the RF module is powered from the 3.3V supply from the LPCXpresso or mbed board. This supply is somewhat current limited (about 100-150mA) but will be sufficient for most RF modules. However, some modules have higher current requirements and then the shorting jumper should be placed in 2-3 position of J12. This is the left position for the jumper, when viewed like in Figure 64. In this case, an external power supply can power the RF module (via U1 voltage regulator).

The RF-module experiments are based on the interrupt-driven UART code from Lab14c. The `UARTSendString()` and `UARTSendBuffer()` functions are needed.

7.16.1 Lab 15a: XBee™ RF-Module

This exercise requires at least two XBee modules and the same number of experiment boards to be able to test XBee communication. One of the boards will be setup as a controller and the other as node(s).

There are many different XBee modules with different functionality and programming interfaces. The XBee module used in this experiment is: XB24-AWI-001. It can be bought from for example Digikey: XB24-AWI-001-ND or Mouser: 888-XB24-AWI-001.

Prepare the boards by plugging in the XBee module in the J15 socket and inserting a jumper in J12 at position 1-2 to get power from the LPCXpresso board.

The XBee driver is quite large and is found in the `xbee.c` and `xbee.h` files in the code framework that is provided. Copy all drivers created so far into the project, including the interrupt driven UART code from Lab 14c.

Below is the template for the `main()` program.

```
#include "stdio.h"
#include "LPC11xx.h"
#include "type.h"
#include "board.h"
#include "gpio.h"
#include "delay.h"
#include "xbee.h"

/*
 * Application configuration
 *
 * CFG_ACT_AS_COORDINATOR - (1) - Configure the XBee module to act as a coordinator.
 *                          (0) - Configure the XBee module to act as an end-device
 */
#define CFG_ACT_AS_COORDINATOR (0)

/*
 * RF message IDs, add your own here but make sure that the coordinator
 * and the nodes have the same numbering.
 */
#define RFPT_SET_LED (1)

// Forward declarations
static void xbeeUp(uint8_t up);
static void xbeeNode(uint32_t addrHi, uint32_t addrLo, uint8_t rssi);
static void xbeeTxStatus(uint8_t frameID, xbeeTxStatus_t error);
static void xbeeData(uint32_t addrHi, uint32_t addrLo, uint8_t rssi,
                    uint8_t* buf, uint8_t len);

static xbee_callb_t callbacks = {
    xbeeUp,
    xbeeNode,
    xbeeTxStatus,
    xbeeData
};

static uint8_t devIsReady = 0;
```

```

volatile uint32_t ms_ticks = 0;

/*****
** Function name: SysTick_Handler
**
** Descriptions: Interrupt handler. Updates the ms_ticks variable to hold
**               the number of milliseconds since start. This will be
**               reasonably accurate and is used by the Xbee driver to
**               handle timeouts.
**
** parameters:   None
** Returned value: The time in milliseconds
**
*****/
void SysTick_Handler(void) {
    ms_ticks += 10;
}

/*****
** Function name: xbeeUp
**
** Descriptions: Xbee node up/down callback.
**
** parameters:   up will be 1 if the node is up, 0 if it is down
** Returned value: None
**
*****/
static void xbeeUp(uint8_t up)
{
    printf("RF: Xbee Up (%d)\r\n", up);
    devIsReady = up;
}

/*****
** Function name: xbeeNode
**
** Descriptions: Xbee node discover callback. Will be called as a response
**               to a Xbee node discovery request. All found nodes are
**               reported back one-by-one through this callback.
**
** parameters:   addrHi - upper 32 bits of the 64-bit node address,
**               addrLo - lower 32 bits of the 64-bit node address,
**               rssi - signal strength
** Returned value: None
**
*****/
static void xbeeNode(uint32_t addrHi, uint32_t addrLo, uint8_t rssi)
{
    printf("RF: Node %x:%x rssi=%d\r\n", addrHi, addrLo, rssi);
}

/*****
** Function name: xbeeTxStatus
**
** Descriptions: Transmit status callback. Called as a result of a packet
**               being sent from the Xbee node.
**
** parameters:   frameId - ID of the frame that was sent,
**               status - status of the transmit request
** Returned value: None
**
*****/
static void xbeeTxStatus(uint8_t frameId, xbeeTxStatus_t status)
{
    if (status != XBEE_TX_STAT_OK) {
        printf("RF: [%d] TX failed %d\r\n", frameId, status);
    }
}

/*****

```

```

** Function name:  xbeeTxStatus
**
** Descriptions:  Received data callback. Called when data has been received
**                by the Xbee node.
**
** parameters:    addrHi - upper 32 bits of the 64-bit node address,
**                addrLo - lower 32 bits of the 64-bit node address,
**                rssi   - signal strength,
**                buf    - buffer containing the data,
**                len    - number of received bytes,
** Returned value: None
**
*****/
static void xbeeData(uint32_t addrHi, uint32_t addrLo, uint8_t rssi,
                    uint8_t* buf, uint8_t len)
{
    int i = 0;
    printf("xbeeData %x:%x, rssi=%d, len=%d\r\n", addrHi, addrLo, rssi, len);

    if (len < 1) {
        return;
    }

    switch (buf[0]) {
        // Set LED request. This is a two byte request where the data
        // indicates if the LED should be turned on or off.
        case RFPT_SET_LED:
            if (len > 1) {
                if (buf[1] == 1) {
                    GPIOSetValue(LED1_PORT, LED1_PIN, LED_ON);
                } else {
                    GPIOSetValue(LED1_PORT, LED1_PIN, LED_OFF);
                }
            }
            break;

        default:
            for (i = 0; i < len; i++) {
                if (i > 0 && (i%8) == 0) {
                    printf("\r\n");
                }
                printf("%x ", buf[i]);
            }
            printf("\r\n");
            break;
    }
}

*****/
** Function name:  sendSetLedRequest
**
** Descriptions:  Broadcasts a request over Xbee to set the status of the LED
**
** parameters:    ledOn - should the LED be lit or not
** Returned value: ERR_OK or an error code
**
*****/
static error_t sendSetLedRequest(uint8_t ledOn)
{
    uint8_t data[2];
    uint8_t id = 0;

    data[0] = RFPT_SET_LED;
    data[1] = ledOn;

    return xbee_send(XBEE_ADDRHI_BROADCAST, XBEE_ADDRLO_BROADCAST, data, 2, &id);
}

int main (void)
{
    error_t err;
    uint8_t state = 0;
    uint8_t oldState = !SW_PRESSED;

```

```

//Set LED1 pin as output
...

//Set SW2 button pin as inputs
...

//Use systick to get an interrupt every 10ms
SysTick_Config(SystemCoreClock / 100);

#if (CFG_ACT_AS_COORDINATOR == 1)
    printf("XBee demo - COORDINATOR\r\n");
    err = xbee_init(XBEE_COORDINATOR, &callbacks);
#else
    printf("XBee demo - NODE\r\n");
    err = xbee_init(XBEE_END_DEVICE, &callbacks);
#endif

    if (err != ERR_OK) {
        printf("Failed to initialize Xbee. Error code %d. Aborting...\n", err);
        while (1) {
            // wait forever
        }
    }

    while (1) {
        xbee_task();

        if (devIsReady != 0) {
            // check button state
            state = GPIOGetValue(SW2_PORT, SW2_PIN);

            if (oldState != state) {
                oldState = state;
                printf("Button: %u\r\n", state);

                sendSetLedRequest(state);
            }
        }
    }

    return 0;
}

```

The XBee driver is provided four callbacks during initialization. The callbacks will be called when the driver has completed initialization of the XBee module (`xbeeUp`), when a new node is discovered (`xbeeNode`), a transfer is completed (`xbeeTxStatus`) and when data is received (`xbeeData`). The driver's `xbee_task()` function must be repeatedly called in order for the XBee module to work correctly.

The `CFG_ACT_AS_COORDINATOR` define should be set to 1 for the controller and 0 for the nodes. Look at the implementation of the `xbee_init()` function to see how they are treated differently.

The last thing to note about the program is the `RFPT_SET_LED` command that is sent when a button is pressed. The command is received by another node and is processed in the `xbeeData()` function.

Run the program on both boards and note that pressing the SW2 button on one board lights the LED on the other board.

Suggested improvements:

- Extend the protocol to retrieve the temperature reading from the other board
- Use one board's quadrature encoder to control the other board's 7-segment display
- Read analog values remotely

- If you have access to more than two XBee modules test what happens when they are all powered.
- Change the protocol from broadcast mode to point-to-point communication by adding the target node's address in the xbee_send() function call.

7.16.2 Lab 15b: GPS Receiver

The Global Positioning System (GPS for short) is a satellite navigation system that provides time and location information as long as there is a direct line of sight to at least four satellites. GPS is used in a wide range of application including cell phones and car navigation systems.

In this experiment a GPS module from Embedded Artists (with the GPS chip from GlobalTop Technology Inc) will be used. It is simple to use as there is no initialization of the module and it continuously sends the received information on the UART channel explored in Lab14a-c. The baud rate is specified by the module to 9600bps.

Note that the GPS module is not included in the component kit. It must be bought separately.

Figure 65 illustrates the GPS module mounted in RF module interface connector J15. The current consumption is low for the module (in the region of a couple of mA) so the shorting jumper can be placed in position 1-2 on J12 (right position in picture below).

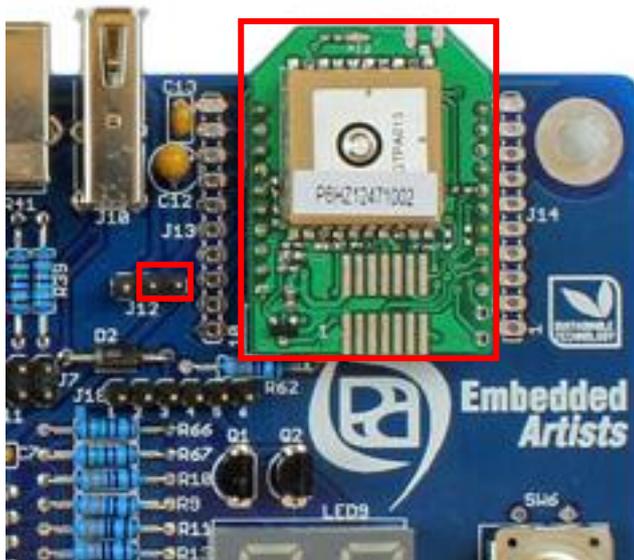


Figure 65 – GPS Module Mounted in J15

The module outputs a number of different messages in the NMEA 0183 format (http://en.wikipedia.org/wiki/NMEA_0183). Each message starts with a dollar sign \$ and ends with a checksum. These are some examples taken from the manufacturer's data sheet:

```
$GPGGA,064951.000,2307.1256,N,12016.4438,E,1,8,0.95,39.9,M,17.8,M,,*65
$GPGSA,A,3,29,21,26,15,18,09,06,10,,,,,2.32,0.95,2.11*00
$GPGSV,3,1,09,29,36,029,42,21,46,314,43,26,44,020,43,15,21,321,39*7D
$GPRMC,064951.000,A,2307.1256,N,12016.4438,E,0.03,165.48,260406,3.05,W,A*2C
```

The exact meaning of each of them is found in the data sheet but here we will focus on the one starting with \$GPGGA as it contains the time and location information.

\$GPGGA,064951.000,2307.1256,N,12016.4438,E,1,8,0.95,39.9,M,17.8,M,,*65

Table-2: GGA Data Format			
Name	Example	Units	Description
Message ID	\$GPGGA		GGA protocol header
UTC Time	064951.000		hhmmss.sss
Latitude	2307.1256		ddmm.mmmm
N/S Indicator	N		N=north or S=south
Longitude	12016.4438		dddmm.mmmm
E/W Indicator	E		E=east or W=west
Position Indicator	Fix 1		See Table-3
Satellites Used	8		Range 0 to 14
HDOP	0.95		Horizontal Dilution of Precision
MSL Altitude	39.9	meters	Antenna Altitude above/below mean-sea-level
Units	M	meters	Units of antenna altitude
Geoidal Separation	17.8	meters	
Units	M	meters	Units of geoids separation
Age of Diff. Corr.		second	Null fields when DGPS is not used
Checksum	*65		
<CR> <LF>			End of message termination

Table-3: Position Fix Indicator	
Value	Description
0	Fix not available
1	GPS fix
2	Differential GPS fix

Figure 66 – GPS Module Data Format

The code below will read one message at a time from the GPS and then extract the time and latitude parts into the gpsData structure.

```
#include "LPC11xx.h"
#include "uart.h"
#include "gps.h" //put typedef declaration below (gpsData) in gps.h file

/**
 * Data structure for the GPS values
 */
typedef struct gpsData {
    uint8_t satellitesUsed[20];
    uint8_t utcTime[20];
    uint8_t altitude[20];
    uint8_t bufLatitude[20];
    uint8_t bufLongitude[20];
    int positionFixed;
    int northSouthIndicator;
    int eastWestIndicator;
    int latitude;
    int longitude;
} gpsData;

static uint8_t END_OF_MESSAGE = '\0';
static uint8_t DIVIDER = ',';

// The parsed data
static gpsData data;
```

```

/*****
** Function name:          hasPattern
**
** Descriptions:          Tests if pBuf starts with pPattern.
**
** parameters:            Buffer to search and pattern to match
** Returned value:        1 if pBuf starts with pPattern, 0 otherwise
**
*****/
static uint8_t hasPattern(uint8_t *pBuf, uint8_t *pPattern)
{
    while(*pBuf != END_OF_MESSAGE && *pPattern != END_OF_MESSAGE){
        if(*pBuf != *pPattern){
            return 0;
        }
        pPattern++;
        pBuf++;
    }
    return 1;
}

/*****
** Function name:          pointToNextValue
**
** Descriptions:          Moves past the next divider
**
** parameters:            Pointer to the string to search
** Returned value:        None
**
*****/
static void pointToNextValue(uint8_t **ppBuf)
{
    while(**ppBuf != END_OF_MESSAGE) {
        if (**ppBuf == DIVIDER) {
            (*ppBuf)++; // point to the start of next value
            break;
        }
        (*ppBuf)++;
    }
}

/*****
** Function name:          convertCordinateToDegree
**
** Descriptions:          Converts the pBuf string which is in the
**                        "ddmm.mmmm" format into an integer representation
**
** parameters:            The buffer, the resulting integer and the
**                        length of the buffer
** Returned value:        None
**
*****/
static void convertCordinateToDegree(uint8_t *pBuf, int* pDegree, int len)
{
    int index = 0;
    int sum = 0;
    int deg = 0;
    int min = 0;
    int div = 0;
    int pow = 1;

    for (index = len; index >=0; index--) {
        if (pBuf[index] == '.') {
            div = 1;
            continue;
        }

        sum += pow * (pBuf[index] & 0x0F);

        if (index > 0) {
            pow *= 10;
            div *= 10;
        }
    }
}

```

```

    div = pow / div;
    deg = sum / (div*100);
    min = sum - (deg*div*100);

    // convert to decimal minutes
    min = (min * 100) / 60;
    *pDegree = (deg*div*100) + min;

    if (div > 10000) {
        // normalize minutes to 6 decimal places
        *pDegree /= (div / 10000);
    }
}

/*****
** Function name:          parseUTC
**
** Descriptions:         Extracts the UTC time string in hhhmmss.sss,
**                       ignoring the .sss part and stores the result
**                       as a string in data.utcTime.
**
** parameters:           The buffer
** Returned value:       None
**
*****/
static void parseUTC(uint8_t **ppBuf)
{
    int index = 0;

    // parse utc hhhmmss.sss
    while(**ppBuf != END_OF_MESSAGE) {
        if(**ppBuf == '.') {
            pointToNextValue(ppBuf);
            break; //reached end of the value
        }

        data.utcTime[index++] = **ppBuf;

        if(index == 2 || index == 5) {
            //Add divider
            data.utcTime[index++] = ':';
        }
        (*ppBuf)++;
    }
    data.utcTime[index] = '\0';
}

/*****
** Function name:          parseLatitude
**
** Descriptions:         Extracts the latitude information and stores
**                       the result as an integer in data.latitude.
**
** parameters:           The buffer
** Returned value:       None
**
*****/
static void parseLatitude(uint8_t **ppBuf)
{
    int index = 0;

    while(**ppBuf != END_OF_MESSAGE) {
        if (**ppBuf == DIVIDER) {
            (*ppBuf)++; //reached end of the value
            break;
        }
        data.bufLatitude[index++] = **ppBuf;
        (*ppBuf)++;
    }
    convertCoordinateToDegree((uint8_t *) &data.bufLatitude, &data.latitude, 8);
}

/*****
** Function name:          GPSRetrieveData

```

```

**
** Descriptions:                Reads and parses the next set of GPS data.
**
** parameters:                 None
** Returned value:            The parsed information
**
*****/
const gpsData* GPSRetreiveData(void)
{
    uint8_t * pattern = (uint8_t*)"GPGGA";

    while (1) {
        uint8_t buf[100];
        uint8_t ch = 0;
        uint8_t *ptr = 0;
        int index = 0;

        // Retrieve the first byte
        if (!UARTGetChar(&ch))
            continue;

        // look for "$GPGGA," header
        if (ch != '$') {
            continue;
        }

        // Retrieve the next six bytes
        for (index=0; index<6; index++) {
            buf[index] = UARTGetCharBlock();
        }

        //Check if its Global Positioning System fixed Data
        if (hasPattern((uint8_t*)&buf, pattern) == 0) {
            continue;
        }

        //Retrieve the data from the GPS module
        for (index=0; index<100; index++) {
            buf[index] = UARTGetCharBlock();

            if (buf[index] == '\r') {
                buf[index] = END_OF_MESSAGE;
                break;
            }
        }

        ptr = &buf[0];

        //parse UTC time
        parseUTC(&ptr);

        //parse Latitude
        parseLatitude(&ptr);

        break;
    }
    return &data;
}

int main (void)
{
    //Set LED1-LED8 pins as outputs
    ...

    //Set SW2/SW3 pins as inputs
    ...

    //initialize the UART to 9600bps 8N1
    UARTInit(9600);

    printf((uint8_t*)"\nWaiting for GPS data...");
}

```

```
//enter forever loop -
while(1)
{
    const gpsData* pData = GPSRetrieveData();
    displayGpsData(pData);
    delayMS(1000);
}

return 0;
}
```

Base the program on the UART functionality developed in Lab14c and place the GPS related code into **gps.c** and **gps.h**.

Run the program to see the current time and latitude. The latitude settings requires at least four satellites, so if the latitude remains 0 after a minute then move closer to a window or perhaps take the board outside.

Extend the program by implementing the functions to at least extract longitude and number of satellites.

Verify your result by entering the coordinates in for example Google Maps or <http://www.findlatitudeandlongitude.com/find-address-from-latitude-and-longitude/>

7.17 Extra: Work with Serial Expansion Connector

In this experiment you will learn how to work with the Serial Expansion Connector. It is a 14-pin connector with SPI, UART and I2C communication interfaces and a couple of GPIOs. The purpose of the connector is to provide a simple expansion connector for smaller expansion modules. Such modules are typically sensors of different kinds and communication modules, but can also be smaller displays. Figure 67 illustrates the interface in the schematics.

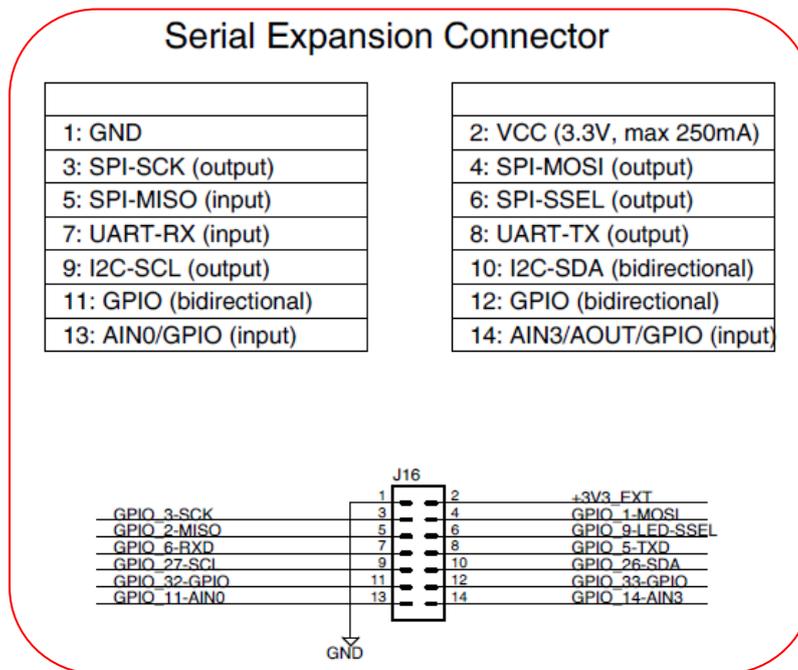


Figure 67 – Serial Expansion Connector on Schematic Page 7

In preparation for the exercise, define all the pins in the Serial Expansion Connector (SEC) connector like below in file **board.h**.

```
#define SEC14_PIN3_PORT    PORT2
#define SEC14_PIN3_PIN    11

#define SEC14_PIN4_PORT    PORT0
#define SEC14_PIN4_PIN    9

//continue with the rest of the pins
...
```

7.17.1 Lab 16a: 128x128 OLED Graphical Display

In this exercise the serial expansion connector will be used to interface a 1.5 inch RGB OLED with a resolution of 128x128 pixels, see product page: http://www.embeddedartists.com/products/displays/15_rgb_oled.php. The display can be bought directly from Embedded Artists, Digkey: EA-LCD-008 or Mouser: 924-EA-LCD-008.

The display contains a built-in controller (SSD1351 from Solomon Systech) that is interfaced via a 4-pin SPI channel (or an 8-bit parallel interface but in this exercise the SPI interface will be used). Pin 3, 4, 6 and 11 are used for the SPI interface. Pin 11 is the fourth signal that is used to differentiate between command and pixel data information transfers. Further, pin 12 is used for reset of the display.

The display module has five DIP switches. All switches except for “pos 3” shall be in “ON” position. See the display module’s schematics for details.

Figure 68 illustrates how the display is connected to the LPCXpresso Experiment board via a 14-pos cable.

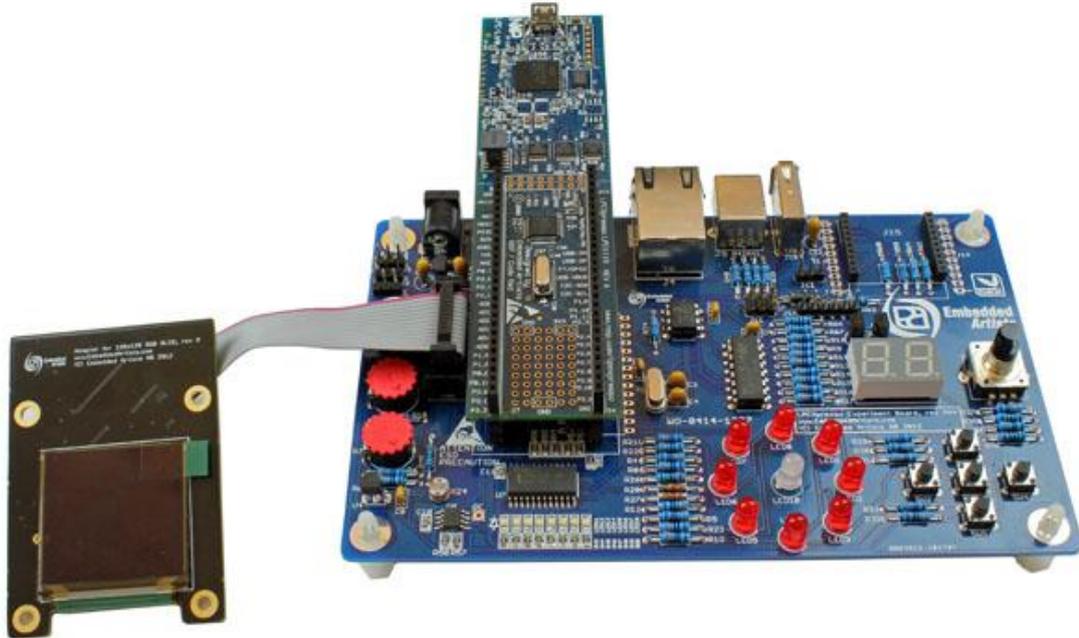


Figure 68 – 1.5 inch RGB OLED Connected via Serial Expansion Connector

In the preparation a number of defines (**SEC14_PIN***) were defined. Each module that is connected via the serial expansion connector can setup its own list of pins depending on needs. For the OLED module it will be:

```
#define OLED_SSEL_PORT    SEC14_PIN6_PORT
#define OLED_SSEL_PIN     SEC14_PIN6_PIN
#define OLED_RESET_PORT  SEC14_PIN12_PORT
#define OLED_RESET_PIN   SEC14_PIN12_PIN
#define OLED_DC_PORT     SEC14_PIN11_PORT
#define OLED_DC_PIN      SEC14_PIN11_PIN
#define OLED_SD_PORT     SEC14_PIN14_PORT
#define OLED_SD_PIN      SEC14_PIN14_PIN
```

The advantage of this approach is that if the LPCXpresso LPC111x board is replaced with a different one then the pin/port information only has to be changed for the **SEC14_*** defines – the connected modules will remain unchanged.

The SSD1315 controller chip is complex and creating a driver from scratch is out-of-scope for this exercise. Instead a number of ready drivers are give, see list below. These files must be copied/imported into the project.

- **draw.c/h** – basic graphical drawing primitives
- **oled.c/h** – OLED initialization function
- **ssd1315.c/h** – OLED controller driver

Below a code segment is given that serves as base for the program in this experiment.

```
static void rainbow(draw_lcd_t *lcd)
{
    // White => 0~15
    draw_fillRectangle(lcd, 0, 0, 15, 127, 0xffff);

    // Yellow => 16~31
    ...
}
```

```

// Purple => 32~47
...

// Cyan => 48~63
...

// Red => 64~79
...

// Green => 80~95
...

// Blue => 96~111
...

// Black => 112~127
...
}

int main (void)
{
    draw_lcd_t lcd;

    // outputs
    GPIOSetDir(OLED_SSEL_PORT,    OLED_SSEL_PIN,    GPIO_OUTPUT);
    GPIOSetDir(OLED_DC_PORT,     OLED_DC_PIN,     GPIO_OUTPUT);
    GPIOSetDir(OLED_RESET_PORT,  OLED_RESET_PIN, GPIO_OUTPUT);

    GPIOSetValue(OLED_SSEL_PORT,  OLED_SSEL_PIN,  1);
    GPIOSetValue(OLED_DC_PORT,    OLED_DC_PIN,    1);
    GPIOSetValue(OLED_RESET_PORT, OLED_RESET_PIN,  1);

    SSP0Init();

    printf("\nInitializing oled driver...");
    oled_init(&lcd);

    rainbow(&lcd);

    //enter forever loop -
    while (1)
        ;

    return 0;
}

```

Run the program and verify that a white bar is shown on the display. Now complete the rainbow function to show 8 differently colored bars on the display.

Explore the other drawing primitives in the **draw.c/h** file.

Some possible improvements of the code base:

- Increase the speed of the SPI bus to 6MHz (originally 1.5MHz)
- Test the **ssd1351_fadeIn()**, **ssd1351_fadeOut()**, **ssd1351_verticalScroll()**, **ssd1351_horizontalScroll()**, **ssd1351_deactivateScroll()** functions. Examples of usage can be found in the software package that came with the OLED display module

7.18 Extra: Work with USB Device

In this experiment you will learn how to work with an USB device interface. This experiment requires an LPCXpresso board with USB device interface. The LPCXpresso Experiment Kit pcb has been designed for the LPCXpresso LPC1769 board, but it is also possible to use the LPC1347 and LPC11U14 boards.

7.18.1 Lab 17a: USB Device – HID

The experiment will configure the LPC1769 as a USB Device with the HID (Human Interface Device) class. The HID driver is always present in Windows and does not require any additional device drivers.

When using the USB Device interface a USB B to USB A cable is needed (not included in the kit). Connect it to the J9 connector on the experiment board and to the PC.

This experiment is based on the code examples that are delivered with the LPCXpresso IDE. The code is structured very differently (none of the code from the previous exercises is used). You have to create a new workspace in LPCXpresso and then import the projects from Lab17a.zip.

Run the USBHID project on the LPCXpresso and let the PC discover it. Run the HIDClient.exe program (found in the USBHID folder of Lab17a.zip) and select LPC17xx USB HID from the list.



Figure 69 – HID Client Screenshot

Press the buttons on the experiment board and note what happens in the PC application. Click the different output checkboxes or change the value in the text field.

The reading/writing of values are done in these two functions in **main.c**:

```

/*
 * Get HID Input Report -> InReport
 */
void GetInReport (void) {
    InReport = 0x00;
    if ((LPC_GPIO0->FIOPIN & (1<<4)) == 0)    InReport |= 0x01; //up    pressed means 0
    if ((LPC_GPIO0->FIOPIN & (1<<2)) == 0)    InReport |= 0x02; //left  pressed means 0
    if ((LPC_GPIO1->FIOPIN & (1UL<<31)) == 0) InReport |= 0x04; //select pressed means 0
    if ((LPC_GPIO0->FIOPIN & (1<<3)) == 0)    InReport |= 0x08; //right  pressed means 0
    if ((LPC_GPIO2->FIOPIN & (1<<7)) == 0)    InReport |= 0x10; //down  pressed means 0
}

/*
 * Set HID Output Report <- OutReport
 */
void SetOutReport (void) {
    static unsigned long led_mask[] = { 1<<6, 1<<17, 1<<15, 1<<16,
                                        1<<3, 1<<13, 1<<28, 1<<27 };
    int i;

    for (i = 0; i < LED_NUM; i++) {
        if (OutReport & (1<<i)) {

```

```

        if (i == 4 || i == 5) LPC_GPIO2->FIOPIN &= ~led_mask[i];
        else                 LPC_GPIO0->FIOPIN &= ~led_mask[i];
    } else {
        if (i == 4 || i == 5) LPC_GPIO2->FIOPIN |= led_mask[i];
        else                 LPC_GPIO0->FIOPIN |= led_mask[i];
    }
}
}

```

Modify the program to use the 7-segment display to show the hexadecimal value from the PC application. Modify the program to read the state of the quadrature encoder to change the value sent to the PC instead of using buttons.

7.18.2 Lab 17b: USB Device – Mouse HID

This experiment uses the USB HID class but this time the USB device will act as a computer mouse. The buttons on the experiment board will control the mouse pointer on the PC.

This experiment is based on the code examples that are delivered with the LPCXpresso IDE. The code is structured very differently (none of the code from the previous exercises is used). You have to create a new workspace in LPCXpresso and then import the projects from Lab17b.zip.

When using the USB Device interface a USB B to USB A cable is needed (not included in the kit). Connect it to the J9 connector on the experiment board and to the PC.

Compile and run the program. Your board should appear as a HID-compatible mouse on the PC when the driver installation has completed.

Use the buttons on the experiment board to move the mouse pointer. The middle button (SW3) acts as a left-click on the mouse.

As seen it is not possible to select text or to move the mouse pointer diagonally. This is because the following code in `main_mouse.c` only allows one button at a time.

The reading/writing of values are done in these two functions in `main_mouse.c`:

```

if (i_JoystickState & JOYSTICK_UP) {
    MouseInputReport.bY = -10;
    MouseInputReport.bX = 0;
    MouseInputReport.bmButtons = 0;
}
else if ((i_JoystickState & JOYSTICK_DOWN)) {
    MouseInputReport.bY = 10;
    MouseInputReport.bX = 0;
    MouseInputReport.bmButtons = 0;
}
else if ((i_JoystickState & JOYSTICK_LEFT)) {
    MouseInputReport.bX = -10;
    MouseInputReport.bY = 0;
    MouseInputReport.bmButtons = 0;
}
else if ((i_JoystickState & JOYSTICK_RIGHT)) {
    MouseInputReport.bX = 10;
    MouseInputReport.bY = 0;
    MouseInputReport.bmButtons = 0;
}
else if ((i_JoystickState & JOYSTICK_CLICK)) {
    MouseInputReport.bX = 0;
    MouseInputReport.bY = 0;
    MouseInputReport.bmButtons = 1;
}
else {
    MouseInputReport.bX = 0;
    MouseInputReport.bY = 0;
    MouseInputReport.bmButtons = 0;
}

```

Fix the code and verify that the mouse pointer can be moved diagonally and that text selection works.

7.19 Extra: Work with USB Host

In this experiment you will learn how to work with an USB host interface. This experiment requires the LPCXpresso LPC1769 board, which has an USB Host interface.

7.19.1 Lab 18a: USB Host

Using the USB Host interface of the LPC1769 it is possible to read/write from a USB Memory Stick. This has a wide range of uses, including

- Providing files for a web server
- Data logging
- Storing of initialization data

The data on the memory stick is persistent, allowing states and data to be kept between power cycles.

This experiment is based on the USBHostLite project that is part of the software package distributed with the LPCXpresso IDE. It expects an **msread.txt** file (content not important) to be present in the root of the memory stick's file system. The file will be copied into (possibly overwriting) **mswrite.txt**. The memory stick should be formatted as FAT.

Note that an external +5V supply is needed, either via J1 or J17. Also note that two jumpers shall be inserted in J11, pos 1-2 and 3-4.

This experiment is based on the code examples that are delivered with the LPCXpresso IDE. The code is structured very differently (none of the code from the previous exercises is used). You have to create a new workspace in LPCXpresso and then import the projects from Lab18a.zip.

Insert a USB memory stick in the J10 connector and then start the program. The program uses semihosting so all printouts will be available in the LPCXpresso IDE. If the memory stick is found and the file is copied the printouts should look like this:

```
Initializing Host Stack
Host Initialized
Connect a Mass Storage device
Mass Storage device connected
Copying from MSREAD.TXT to MSWRITE.TXT...
Copy completed
```

If the memory stick was not inserted before the program started it will look like this:

```
Initializing Host Stack
Host Initialized
Connect a Mass Storage device
ERROR: In Host_EnumDev at Line 407 - rc = -1
```

This is not very user friendly. Improve the implementation of **main()** to wait for the memory stick to be inserted, copy the file, wait for the memory stick to be removed and then start over.

7.20 Extra: Work with Ethernet Interface

In this experiment you will learn how to work with the Ethernet interface and TCP/IP. This experiment requires the LPCXpresso LPC1769 board, which has an Ethernet interface.

7.20.1 Lab 19a: easyWeb Web Server

This experiment will demonstrate a very basic web server that is a part of the software package distributed with the LPCXpresso IDE. The web server returns the same web page regardless of which URL is requested. For example <http://192.168.5.200> returns the same page as http://192.168.5.200/test/a_page.html.

This experiment is based on the code examples that are delivered with the LPCXpresso IDE. The code is structured very differently (none of the code from the previous exercises is used). You have to create a new workspace in LPCXpresso and then import the projects from Lab19a.zip.

Connect an Ethernet cable between the J4 connector on the experiment board and a network hub/switch/router depending on the network you are connected to.

The project needs a couple of small changes to work. Start by creating a unique Ethernet address (MAC address) by opening `ethmac.h` and modifying these lines:

```
#define MYMAC_1      1      // our ethernet (MAC) address
#define MYMAC_2      2      // (MUST be unique in LAN!)
#define MYMAC_3      3
#define MYMAC_4      4
#define MYMAC_5      5
#define MYMAC_6      8
```

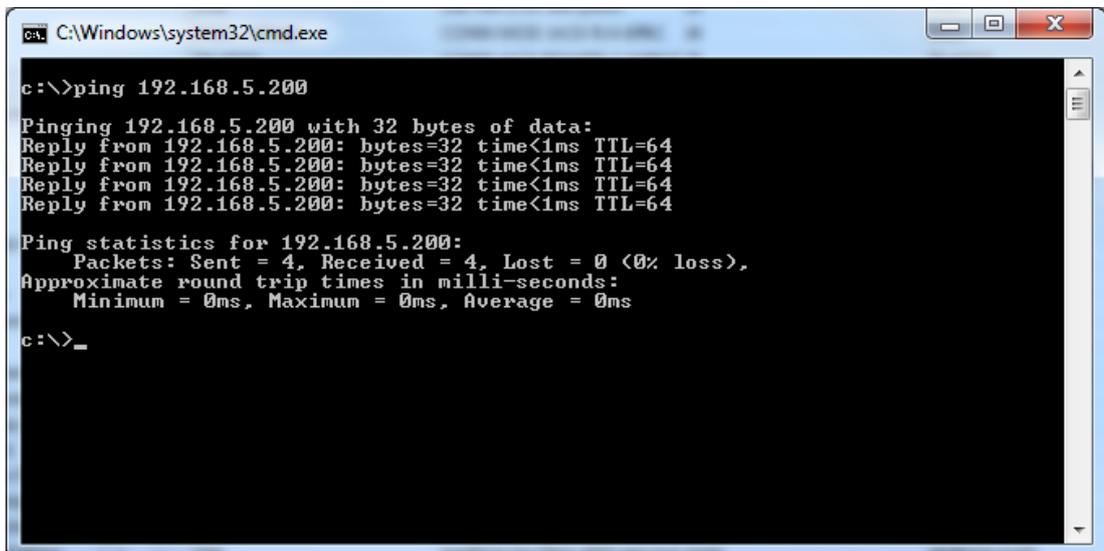
The address must be unique on the network. If you are doing these labs with other students/colleagues then you must all agree on which addresses to use to avoid name conflicts.

After selecting an Ethernet address it is time to select a fixed IP address. This is set in `tcip.h` like this:

```
#define MYIP_1      192    // our internet protocol (IP) address
#define MYIP_2      168
#define MYIP_3      5
#define MYIP_4      200
```

This address also has to be unique within your network.

Compile and run the code. To see if everything works, open a command prompt and run `ping` to see if your board responds (make sure to use the ip address you entered in `tcip.h`):



```
C:\Windows\system32\cmd.exe

c:\>ping 192.168.5.200

Pinging 192.168.5.200 with 32 bytes of data:
Reply from 192.168.5.200: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.5.200:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

c:\>_
```

Figure 70 – PING Screenshot

If you get replies as shown above everything is working. If not then go back to `ethmac.h` and `tcPIP.h` and verify that the addresses you have selected are correct.

Now open a web browser and enter your selected IP number in the address field. You should get a page similar to this.

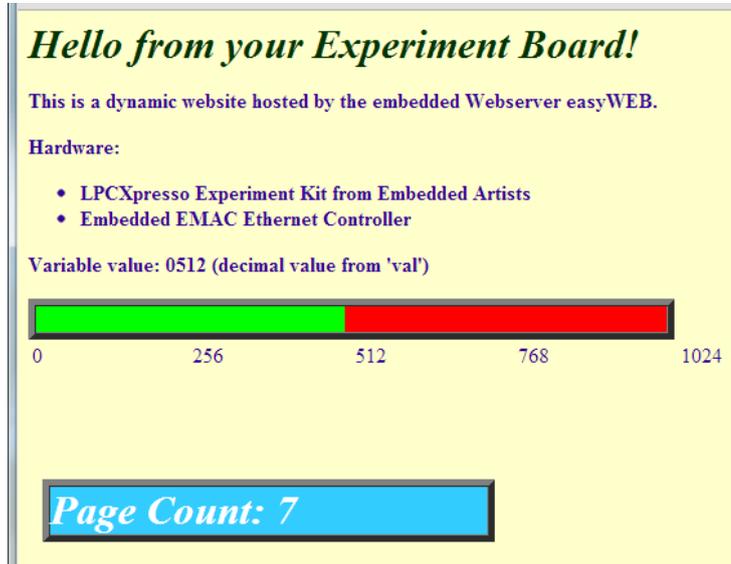


Figure 71 – Web Page Screenshot

The page is continuously updated - page count is increasing and the colored bar is changing value. This is accomplished by having a static web page with a couple of fields that are updated before the page is sent to the browser. The static page is declared in `webside.h`:

```
const unsigned char WebSide[] = {
  "<html>\r\n"
  "<head>\r\n"
  "<meta http-equiv=\"refresh\" content=\"1\">\r\n"
  "<title>easyWEB - dynamic Webside</title>\r\n"
  "</head>\r\n"
  "\r\n"
  ...
}
```

The dynamic content comes from the `InsertDynamicValues()` function in `easyweb.c`. It locates and replaces the markers in the `WebSide[]` data.

Suggested changes:

- Change colors on the page
- Make the colored bar decrease instead of increase
- Add a second bar and let the two bars represent the values on the two trimming potentiometers.
- Read and present the current temperature

7.20.2 Lab 19b: lwIP TCP/IP Stack, Web Server and FreeRTOS

- 1) Go to [FreeRTOS+IO and FreeRTOS+CLI demo2](#)
- 2) Download the projects, [LPC1769 FreeRTOS Plus Featured Demo 002.zip](#)
- 3) Create a new workspace and import the contents of the downloaded zip file

- 4) Remove the call to `vStartSPIInterfaceToSDCardTask()` from `main()` in `FreeRTOS-Plus-Demo-2\Source\main.c`. This is needed to prevent the demo from crashing as a result of a missing SD card.
- 5) Modify the `FreeRTOS-Plus-Demo-2\Source\FreeRTOSConfig.h` to get unique addresses:

```

/* MAC address configuration. */
#define configMAC_ADDR0      0x00
#define configMAC_ADDR1      0x12
#define configMAC_ADDR2      0x13
#define configMAC_ADDR3      0x10
#define configMAC_ADDR4      0x15
#define configMAC_ADDR5      0x12

/* IP address configuration. */
#define configIP_ADDR0        192
#define configIP_ADDR1        168
#define configIP_ADDR2        5
#define configIP_ADDR3        201

```

Run the ping test as described in <insert ref here> and then point the web browser to `http://<your_selected_ip_number>`. The page will look like this:

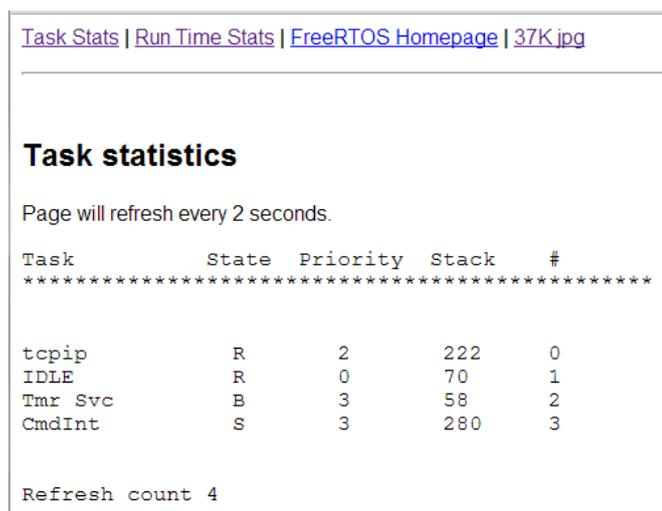
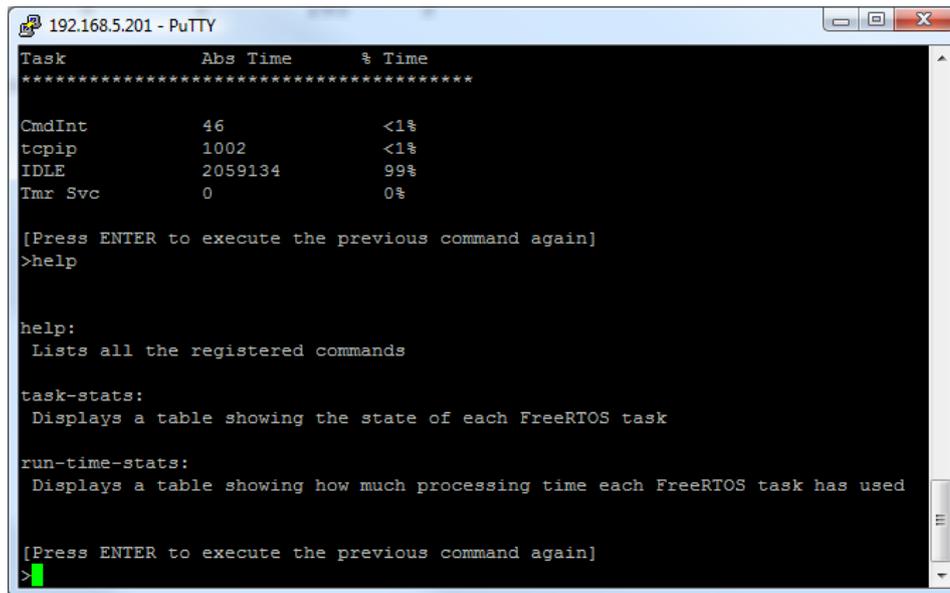


Figure 72 – Task Statistics Screenshot

The FreeRTOS allows multiple threads to run in (seemingly) parallel. The program has both a web server and a telnet server. The telnet server can be accessed by e.g. Putty on port 22.



```
192.168.5.201 - PuTTY
Task      Abs Time    % Time
*****
CmdInt    46          <1%
tcpip     1002        <1%
IDLE     2059134     99%
Tmr Svc   0           0%

[Press ENTER to execute the previous command again]
>help

help:
  Lists all the registered commands

task-stats:
  Displays a table showing the state of each FreeRTOS task

run-time-stats:
  Displays a table showing how much processing time each FreeRTOS task has used

[Press ENTER to execute the previous command again]
>
```

Figure 73 – Telnet Server Screenshot

Test the three available commands: **help**, **task-stats** and **run-time-stats**.

7.21 Differences between LPCXpresso LPC111x and LPC1114 in DIL28

The experiments are based on the LPCXpresso LPC111x boards that can be based on the LPC1114 or the LPC1115. For all practical purposes, these two chips are interchangeable. The LPC1115 has the double amount of FLASH (64kByte instead of 32kByte for the LPC1114). At the time of writing the LPCXpresso LPC1114 board has been discontinued in favor for the LPC1115 version, so the majority of users will likely work with the LPCXpresso LPC1115 board.

One of the included components is the LPC1114FN28/102 chip, which comes in a DIP28 package. That is a package that can easily be used on a breadboard. Figure 74 below (from the LPC111x User's manual) lists the differences.

Type number	Series	Flash	Total SRAM	Power profiles	UART RS-485	I ² C/ Fast+	SPI	ADC channels	GPIO	Package
LPC1113FHN33/203	LPC1100XL	24 kB	4 kB	yes	1	1	2	8	28	HVQFN33
LPC1113FHN33/301	LPC1100	24 kB	8 kB	no	1	1	1	8	28	HVQFN33
LPC1113FHN33/302	LPC1100L	24 kB	8 kB	yes	1	1	1	8	28	HVQFN33
LPC1113FHN33/303	LPC1100XL	24 kB	8 kB	yes	1	1	2	8	28	HVQFN33
LPC1113FBD48/301	LPC1100	24 kB	8 kB	no	1	1	2	8	42	LQFP48
LPC1113FBD48/302	LPC1100L	24 kB	8 kB	yes	1	1	2	8	42	LQFP48
LPC1113FBD48/303	LPC1100XL	24 kB	8 kB	yes	1	1	2	8	42	LQFP48
LPC1114										
LPC1114FDH28/102	LPC1100L	32 kB	4 kB	yes	1	1	1	6	22	TSSOP28
LPC1114FN28/102	LPC1100L	32 kB	4 kB	yes	1	1	1	6	22	DIP28
LPC1114FHN33/201	LPC1100	32 kB	4 kB	no	1	1	1	8	28	HVQFN33
LPC1114FHN33/202	LPC1100L	32 kB	4 kB	yes	1	1	1	8	28	HVQFN33
LPC1114FHN33/203	LPC1100XL	32 kB	4 kB	yes	1	1	2	8	28	HVQFN33
LPC1114FHN33/301	LPC1100	32 kB	8 kB	no	1	1	1	8	28	HVQFN33
LPC1114FHN33/302	LPC1100L	32 kB	8 kB	yes	1	1	1	8	28	HVQFN33
LPC1114FHN33/303	LPC1100XL	32 kB	8 kB	yes	1	1	2	8	28	HVQFN33
LPC1114FHN33/333	LPC1100XL	56 kB	8 kB	yes	1	1	2	8	28	HVQFN33
LPC1114FHI33/302	LPC1100L	32 kB	8 kB	yes	1	1	1	8	28	HVQFN33
LPC1114FHI33/303	LPC1100XL	32 kB	8 kB	yes	1	1	2	8	28	HVQFN33
LPC1114FBD48/301	LPC1100	32 kB	8 kB	no	1	1	2	8	42	LQFP48
LPC1114FBD48/302	LPC1100L	32 kB	8 kB	yes	1	1	2	8	42	LQFP48
LPC1114FBD48/303	LPC1100XL	32 kB	8 kB	yes	1	1	2	8	42	LQFP48
LPC1114FBD48/323	LPC1100XL	48 kB	8 kB	yes	1	1	2	8	42	LQFP48
LPC1114FBD48/333	LPC1100XL	56 kB	8 kB	yes	1	1	2	8	42	LQFP48
LPC1115										
LPC1115FBD48/303	LPC1100XL	64 kB	8 kB	yes	1	1	2	8	42	LQFP48

Figure 74 – LPC111x Variant Comparison

There are some differences when working with the LPC1114FN28/102 chip. Memory-wise, the difference is small. The chip has half the amount of SRAM (4kByte) when compared to the LPC1115/1114 on an LPCXpresso board. The big difference is less available pins. When working on the breadboard the solution is simply to switch which pins to use (since all are not used simultaneously), so no problem there. When having soldered all components to the PCB, the setup is more fixed. Most of the pins that are lacking on the DIP28 package have been routed to U7 (PCA9532), the I2C-GPIO expander. That way it is possible to access these signals (pins) via I2C instead. The experiments affected with pcb mounted components are the following:

- 8 LEDs, signals GPIO_9-LED-SSEL, GPIO_21-LED, GPIO_22-LED and GPIO_23-LED are not connected to the LPC1114FN28. GPIO_9-LED-SSEL is however connected to U7 (PCA9532).

- 5 push-buttons, signals GPIO_17-KEY, GPIO_18-KEY and GPIO_35-KEY are only connected to U7 (PCA9532). The signals can be manually bridged to any other free pin in the specific experiment.
- Buzzer, signal GPIO_7-BUZZ is not connected at all. The signal can be manually bridged to any other free pin in the specific experiment.
- RGB-LED, signal GPIO_30-PWM is only connected to U7 (PCA9532). This is the signal controlling the green LED. Note that signals GPIO_28-PWM and GPIO_29-PWM are both connected to the LPC1114FN28 and U7. This is because U7 has built-in functionality for PWM control so there is a possibility to experiment with this specifically.
- Quadrature encoder, signals GPIO_38-QA and GPIO_39-QB are only connected to U7 (PCA9532). The signals can be manually bridged to any other free pin in the specific experiment.

8 Projects

This chapter contains a list of project ideas that build on the knowledge gained from the experiments in the previous chapters. The projects involve a bigger programming effort than before and are real-world in the sense that part of a real product application can very likely contain one of the project ideas. The idea is to deepen your understanding of embedded systems and enhance your programming skills.

The projects are not described in detail like the experiments. Instead the descriptions are quite short and are mainly supposed to give you some ideas and get you started. Solve the details on your own – that is what programming is all about anyways! Alternatively create your own project based on these ideas.

8.1 Interface a Color Sensor

Select a color sensor, for example one of these:

- <http://www.sparkfun.com/products/10701> (with a digital interface)
- <http://www.sparkfun.com/products/10904> (with an analog interface)

Create the hardware and software interface to the sensor. Output can be on the console, the RGB-LED or on a display.

8.2 Interface a Real-time Clock (RTC)

Select an RTC chip and interface. Most commonly used interfaces to these chips are I2C or SPI. For example NXP PCF8523 with I2C interface or PCF2123 with SPI interface.

Create an application that displays the real-time on a display or via the console. It shall be possible to set the current time. If a display is used, create a small menu system controlled by the joystick push-buttons or the rotary switch.

Implement alarm functionality.

Implement low-power operation where the processor sleep and only wake up once a second to update the time, or even once a minute. The processor shall also wake up on alarms.

Several enhancements are possible to this project:

- Implement automatic adjustment of the clock once a day.
- Implement automatic adjustment for summer and winter time.
- If the RTC chip does not support leap year add support for leap year compensation.

8.3 Interface a GPS Module

Interface a GPS module. Most modules have a UART interface and communicate with the standard NMEA protocol (use google to find more information about this protocol specification).

A simpler project just output the results from the GPS module on a display with X/Y-coordinates. A more advanced project visualizes the location on a graphical display.

8.4 Interface an SD/MMC Memory Card

Interface an sd/mmc memory card via the spi bus. There are application notes from NXP that gives a good start. Add FAT-file system handling “on top” of the low-level interface drivers.

8.5 Interface an Accelerometer and Gyro

Interface an accelerometer (2- or 3-axis) or a gyro. There are several chips with associated breakout boards for simpler interfacing on the market. Select a chip with digital interface (I2C or SPI).

8.6 Control a LED Matrix

Interface an 8x8 LED matrix. There are both single color and RGB-LED matrixes. Create an application that can control each individual LED in the matrix.

To control the matrix it is suggested to have a timer interrupt that updates the LED matrix in a multiplexed way, i.e., one columns or one row at a time. The frequency must typically be at least 100 Hz in order to avoid flickering. The timer interrupt function can get information about which LEDs to turn on/off from a 64 bit array, i.e., a vector of 8 bytes. In case of an RGB-matrix, three such bit arrays are needed, one for each color.

As a start, create an application that updates the 64 LEDs so that messages can be streamed. A simple solution is to just store the bit pattern of the message. A more advanced solution can store the messages as ASCII strings. In the latter case, a bit map defining all characters must also be defined.

For more advanced, and fun, use, create a game for the LED matrix. See for example these projects for some ideas:

- <http://www.evilmadscientist.com/2008/meggy-jr-rgb/>
- <http://hackaday.com/2010/02/19/update-most-interesting-game-in-64-pixels/>
- <http://interactive-matter.eu/blog/2010/05/08/blinken-buttons-for-beginners-a-smt-beginners-kit/>

8.7 Create a Game with Display + Accelerometer or Gyro

Create a game with a graphical display and an accelerometer or gyro. Navigating a rolling ball in a labyrinth or recreate the classical snake game, for example.

8.8 Create General Menu System for a Display

Create a general menu system for small character based LED, for example a 2x20 character display. Use the joystick-buttons or the rotary switch as user input.

To get some ideas about which functions that are needed in a menu system have a look at older (non smart phone) cell phones. These phones had small displays and few buttons. There you can find many typical functions that are needed.

Character based LCDs typically have an 8-bit parallel interface. Many of them also have a 4-bit interface mode (to save interface pins). Write code that works for both modes.

Alternatively use a graphical display for more flexibility and nicer looking interfaces.

8.9 Retrieve Information from Web Servers

A web browser used the HTTP communication protocol (on top of TCP/IP) to retrieve information from web servers. More specifically, it's the GET request that is used.

Create an application that connects to a web server (typically port 80) and send a HTTP GET request and displays the information in a suitable way, for example on the console or a display. When the data is received it must be interpreted in order to extract the usable information.

A typical setup can be a system with a web server that resents the analog values of the two analog inputs on a HTML-page. If another embedded system shall also retrieve this information, it must interpret the HTML-data and extract the correct information, i.e., the analog values.

8.10 USB Mouse Emulation

Create a USB device application that emulates a USB mouse. You need to implement a USB HID device. HID stands for Human Interface Device, which is exactly what a mouse is. Study USB-related documentation to find out more about this.

The joystick switches can be used to move the mouse position, and in the end move the cursor pointer on a PC screen.

8.11 Registry in E2PROM

Create a so called registry, which is a non-volatile storage that can store values connected to so called keys. Non-volatile storage is easily created by using the E2PROM, which is accessible over the I2C-bus. The keys can be short strings, for example strings with lengths between 1 and 16. To make it simple, a value connected to a key is always a 32-bit integer.

If you want to make it more advanced, a value can also be a string.

Typical operations for a registry are:

- Create entry for a new key
- Delete key
- Get value of key
- Update key
- List all keys

A registry is always usable to have when creating real-world applications. There is often a need to store settings from a user. Also in the case of Internet communication the following settings must be stored in the system:

- IP address
- Subnet mask
- Default gateway
- If an Ethernet interface is present, the MAC address is also needed

8.12 Real-Time Dynamic Data with JAVA Applet

This is a project that requires a TCP/IP stack, web server and RTOS.

By using SSI and CGI/EGl technologies (in a web server) you can create dynamic information in a web server. However, the information is created at the download moment. Anything that happens after that point in time will not be reflected on the client side. True real-time data visualization is hence not possible with SSI or CGI/EGl.

To create true real-time data on the client side (= the web browser) a JAVA applet must be used. The JAVA applet connects back to the embedded system, opens a communication channel, and then received real-time data. The communication channel can be either a TCP or a UDP connection.

There must of course be a TCP/UDP server on the system that produces the real-time data. It is this server that the JAVA applet connects to. Create such a system!

Let the server produce a stream of data, for example a sinus-valued signal stream. Alternatively, the server can stream the values from both analog inputs of the board. Let the JAVA applet display this data stream in a suitable way.

8.13 Multiplayer Game via RF-module

Create a multiplayer game with the help of RF-modules. For example ping-pong on a graphical display.

8.14 Home Alarm System

Create a home alarm system with remote sensors that communicate wirelessly with a central. It can even communicate with a GSM/3G phone modem and make a call/send an SMS in case of an alarm.

8.15 Polyphonic Audio Generation

Create an audio output with speaker amplifier. A low-pass filtered PWM-output can be used to generate the audio waveform. Implement polyphonic tone generation and play a melody with multiple tones.

8.16 Audio Processing

Create an audio interface with both input and output. Create audio effects, for example an echo chamber. Alternatively create a system for voice recording and playback.

8.17 Home Automation

Create a system for home automation. There are many different systems that can be created for handling ventilation, heating, blower watering, lighting, etc. It can be something very small or a big system with remote nodes communication wirelessly and presenting information on the Internet.

8.18 Control a Robot

Find a suitable mechanical platform and let the LPC111x control the robot hardware.

8.19 RS-485 Network

Create an RS-485 network and define a protocol for reliable communication, for example a token passing protocol.

8.20 Interface an FPGA/CPLD Chip

Create a project that interface a programmable hardware chip (FPGA or CPLD). There are many small breakout boards on the market with these kinds of chips. Implement some kind of hardware in the programmable chip and then let the microcontroller control this functionality. It can be like an additional peripheral to the microcontroller.

8.21 Analog Electronic Experiments

This is more than just one individual project idea. The experiments and kit content address digital electronics. There is a whole world of analog electronic also, including the interface between the analog world and the digital world. Create own experiments to learn areas of interest, for example:

- How to use a comparator to interface an analog signal to a digital input.
- How to adjust the range of an analog signal before feeding it to an ADC (Analog to Digital Converter), both gain and offset.
- How to apply low-pass filtering before sampling an analog signal. What are the theoretical requirements and practical implications?
- How to create an analog signal via low-pass filtering a PWM signal.
- How to create a capacitive touch sensor.
- How to use an opto-coupler to galvanically isolate a digital signal.

9 LPCXpresso IDE – How to get Started

This chapter gives a quick presentation of how to get started with the LPCXpresso IDE, which is the integrated program development environment that was created for the LPCXpresso board family. There are also more extensive and detailed presentations and descriptions on the LPCXpresso website [5].

Before starting, make sure that the latest version of the LPCXpresso IDE is installed. See [5], <http://www.nxp.com/lpcxpresso/> for details where to download and how to install.

9.1 Importing Projects

A package of projects has been created as a base for supporting the experiments. It makes it easier to get up-and-running quicker with the first set of experiments. Download this package from Embedded Artists support page after registering the product. It is a zip-file that contains all project files and is a simple way to distribute complete Eclipse projects.

This section describes how to import the package of projects into the Eclipse workspace. Several projects will be imported simultaneously.

Start the LPCXpresso IDE and select a new (and empty) workspace directory.

Select the *Import and Export* tab in the Quickstart menu and then *Import archived projects (zip)*, see figure below.

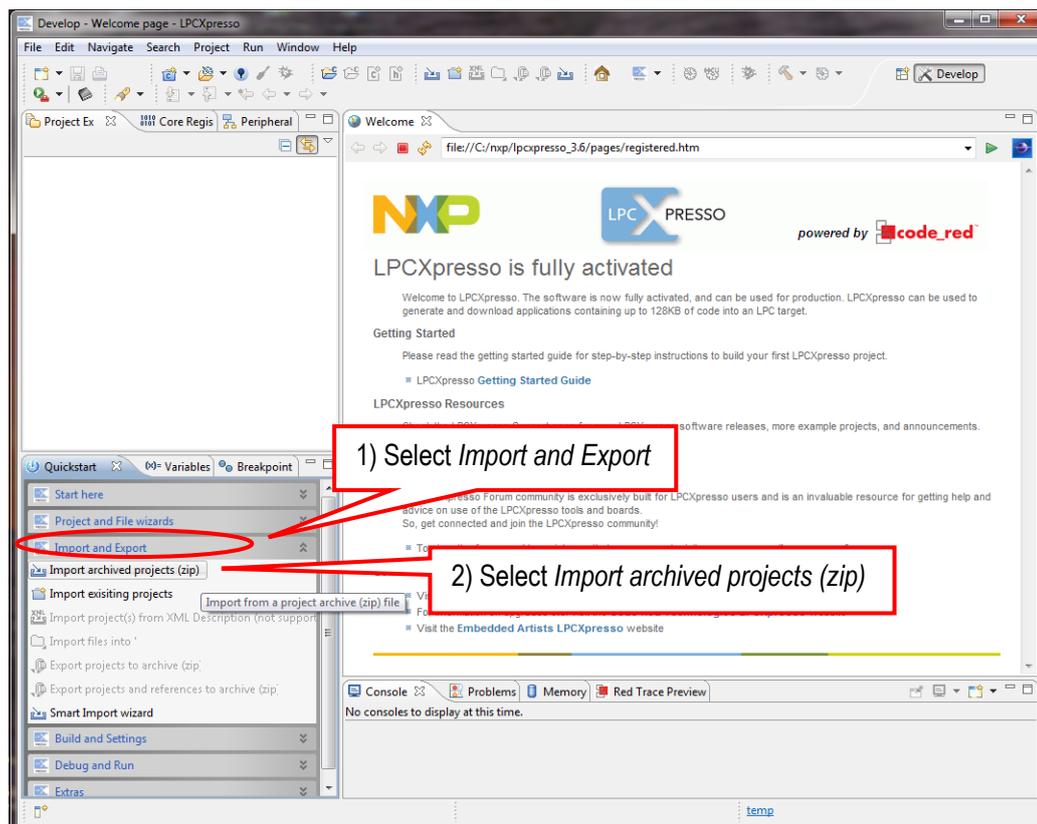


Figure 75 – LPCXpresso IDE Import Archived Project

Next, browse and select the downloaded zip file containing the archived projects. Make sure all sub-projects are selected to be imported, see figure below (note that the screen shot below is generic and the project names will be different).

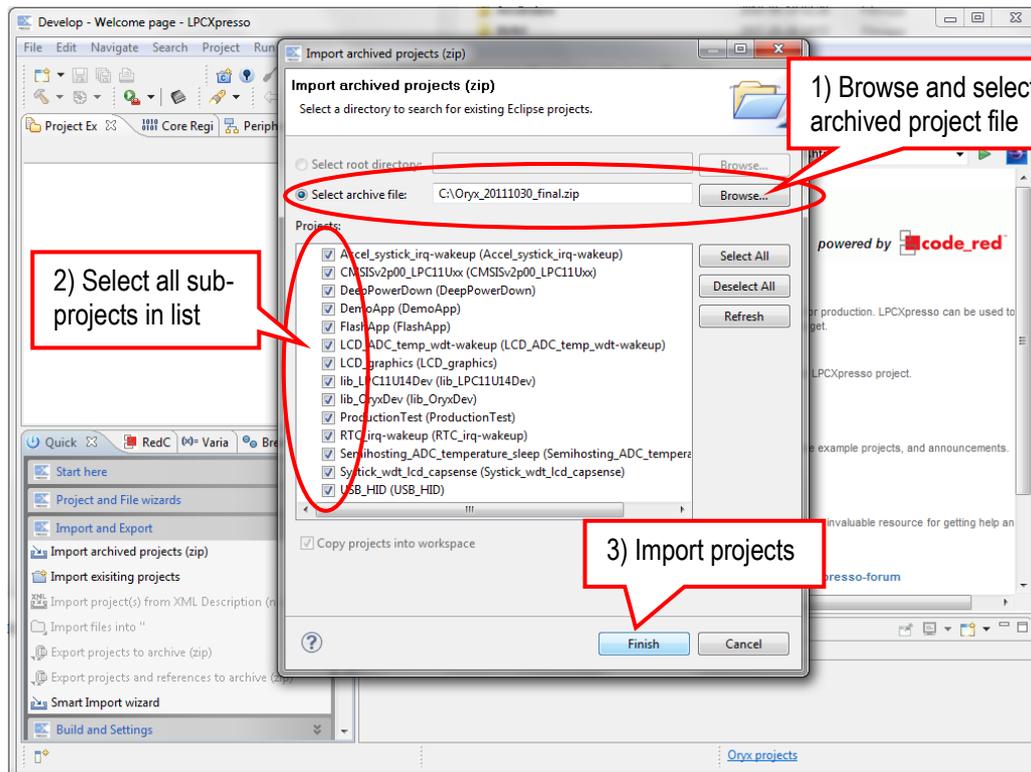


Figure 76 – LPCXpresso IDE Import Archived Project Window

All projects are now imported.

9.2 Working with a Project and Compiling

Click (to select) the project to work with. Note that there are several projects in the workspace.

Browse and edit the project files. These are typically found under the src sub-directly. The main window to the right in the LPCXpresso window is a source code editor.

Build and **clean** the project from the Quickstart menu (*Start here*), see picture below. When compiling, the console window in the lower (right) corner of the LPCXpresso Window will give information about the compile and link process.

When the project compiles and links without any errors it is possible to move to the next step (next section) and download the code to the LPC111x - and begin the debug session.

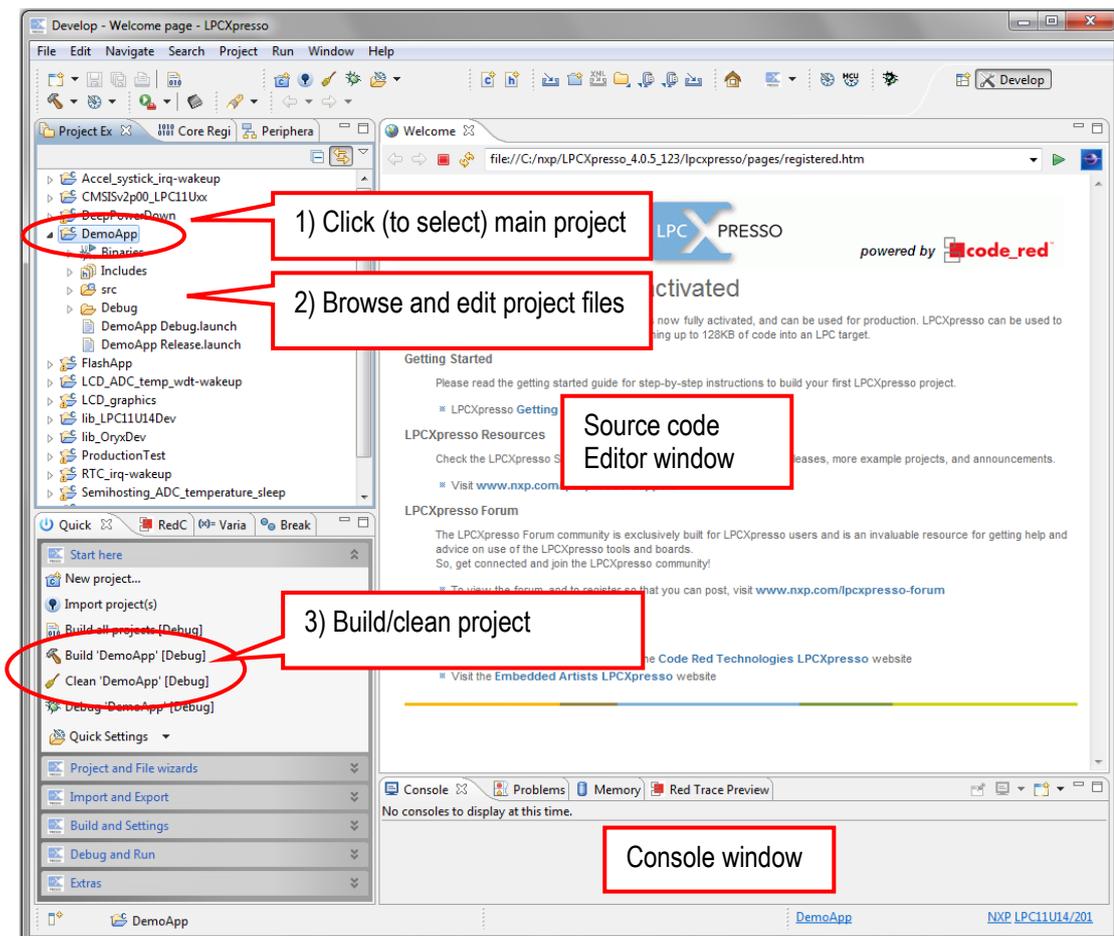


Figure 77 – LPCXpresso IDE Build Project

9.3 Debugging a Project and Downloading

When the project compiles and links without any errors it is time to start debugging – to download the code to the LPC111x and start executing!

Before starting to debug, make sure the LPCXpresso board is connected (via USB) to the PC. The code is downloaded to the board via this cable. The LPCXpresso board consists of two parts, one is the LPC111x processor and the other is the LPC-Link™ side, which is an embedded debug interface.

Click (to select) the project to work with. Click on **debug** in the Quickstart menu (*Start here*), see picture below.

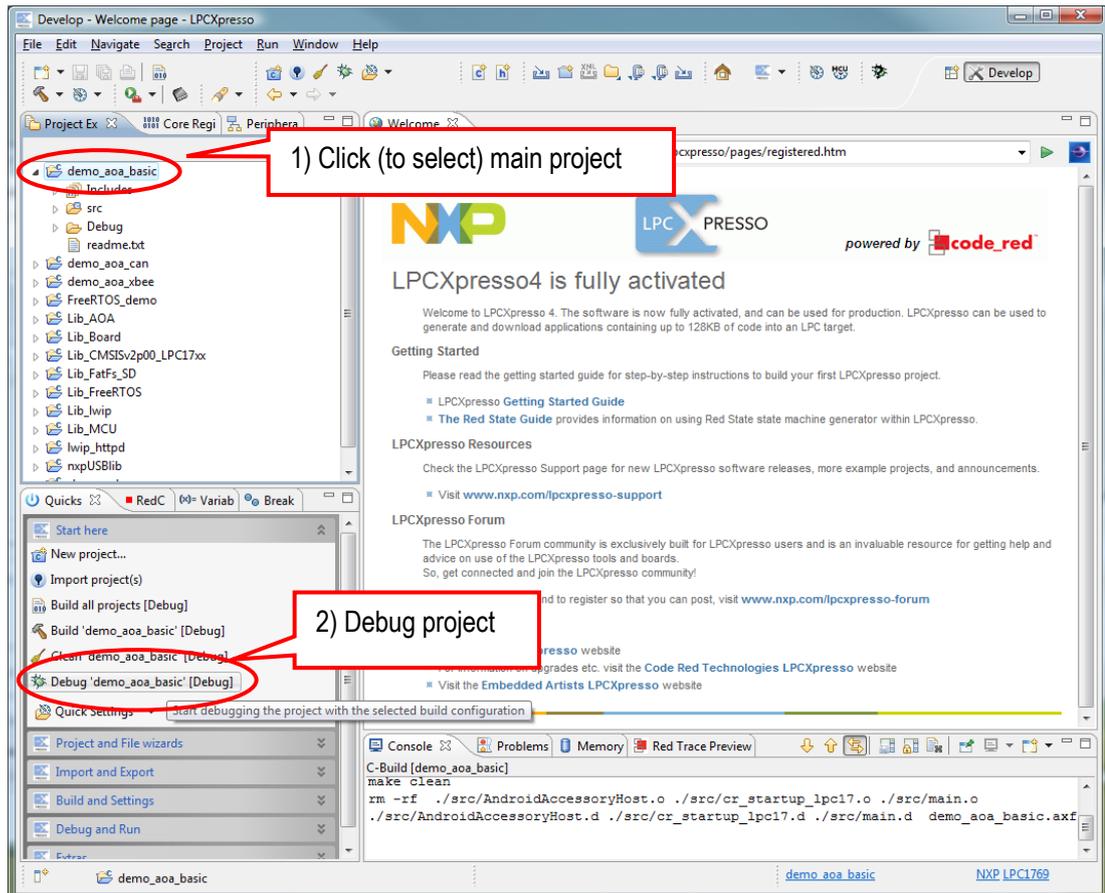


Figure 78 – LPCXpresso IDE Debug Project

In case flashing fails, an error message like below will be displayed. This is an indication that the debugger could not connect to the LPC111x. The most common reason is that the microcontroller is in a low-power mode where the debug connection is disabled. Make sure the microcontroller is in ISP/bootload mode and try again. This is accomplished by pulling pin PIO0_1 low (via 100-1000 ohm resistor to ground).

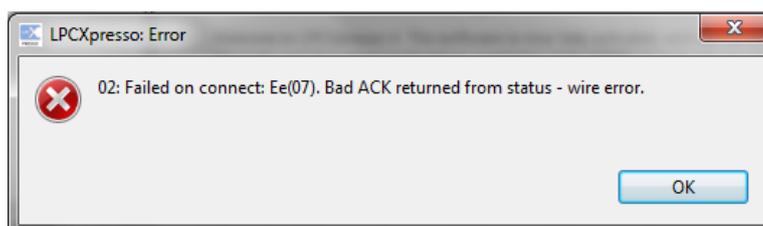


Figure 79 – LPCXpresso IDE Program Failing to Flash

When the code has been downloaded execution will stop at the first line in the main function. Press F8 or the green arrow button to resume/start execution.

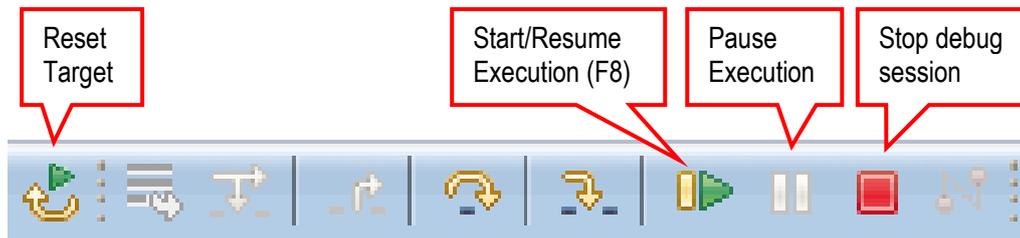


Figure 80 – LPCXpresso IDE Run Button

It is possible to manually stop execution by pressing the **Pause** button. After that it is possible to restart the execution by pressing the **Start** button (or F8). Alternatively the target can be reset by pressing **Reset** button and the system return to the state just after program download, i.e., at the first line in the main function.

The debug session is ended by pressing the **Stop** button. The LPCXpresso IDE then returns to edit mode.

When the system has been stopped the call stack window indicates where the execution has stopped and the call path to get to that point.

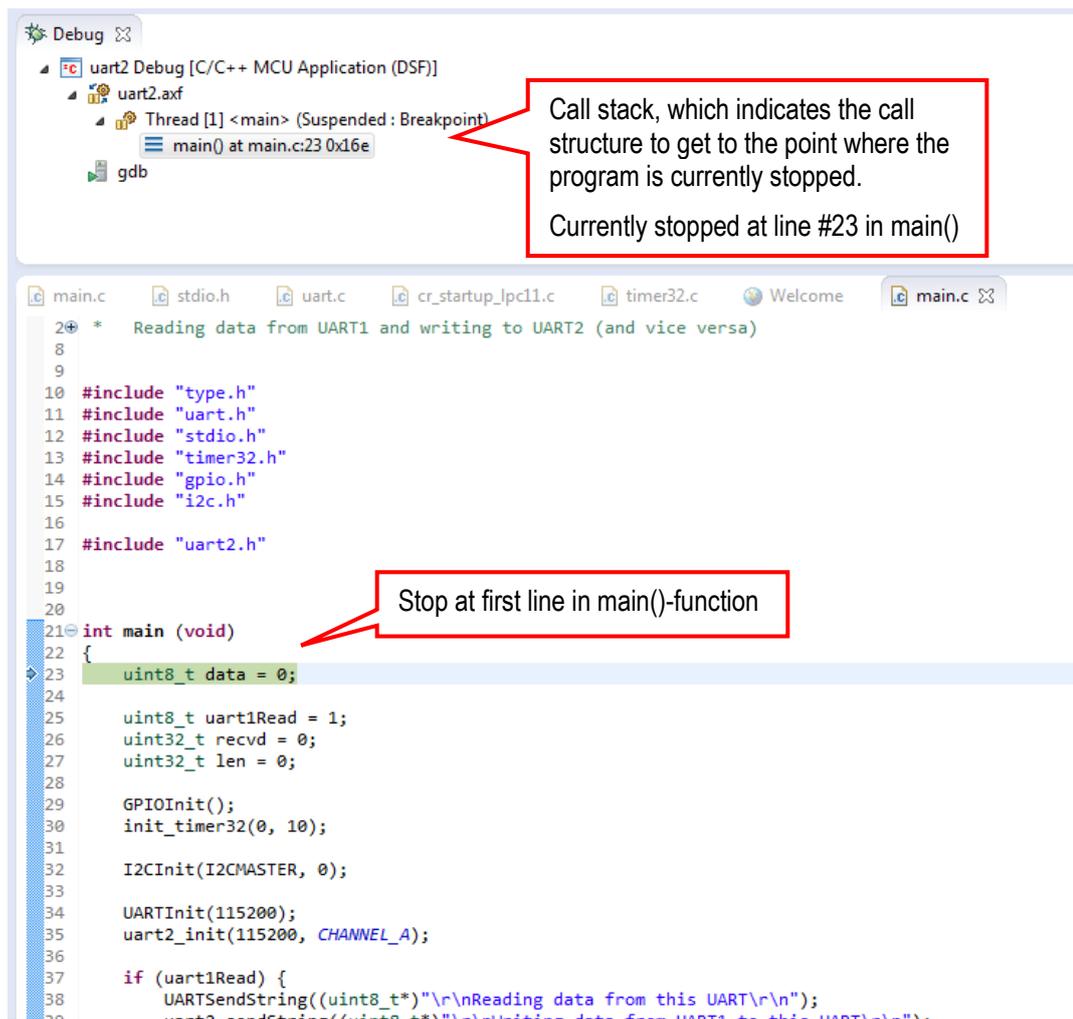


Figure 81 – LPCXpresso IDE Stop at First Line in main()

It is possible to set breakpoints by double clicking in the left margin. A small dot marks that a breakpoint has been set to a specific source code line. In Figure 82 below, the breakpoint has been set to line #34 in function main(). A breakpoint is removed by double-clicking on the dot.

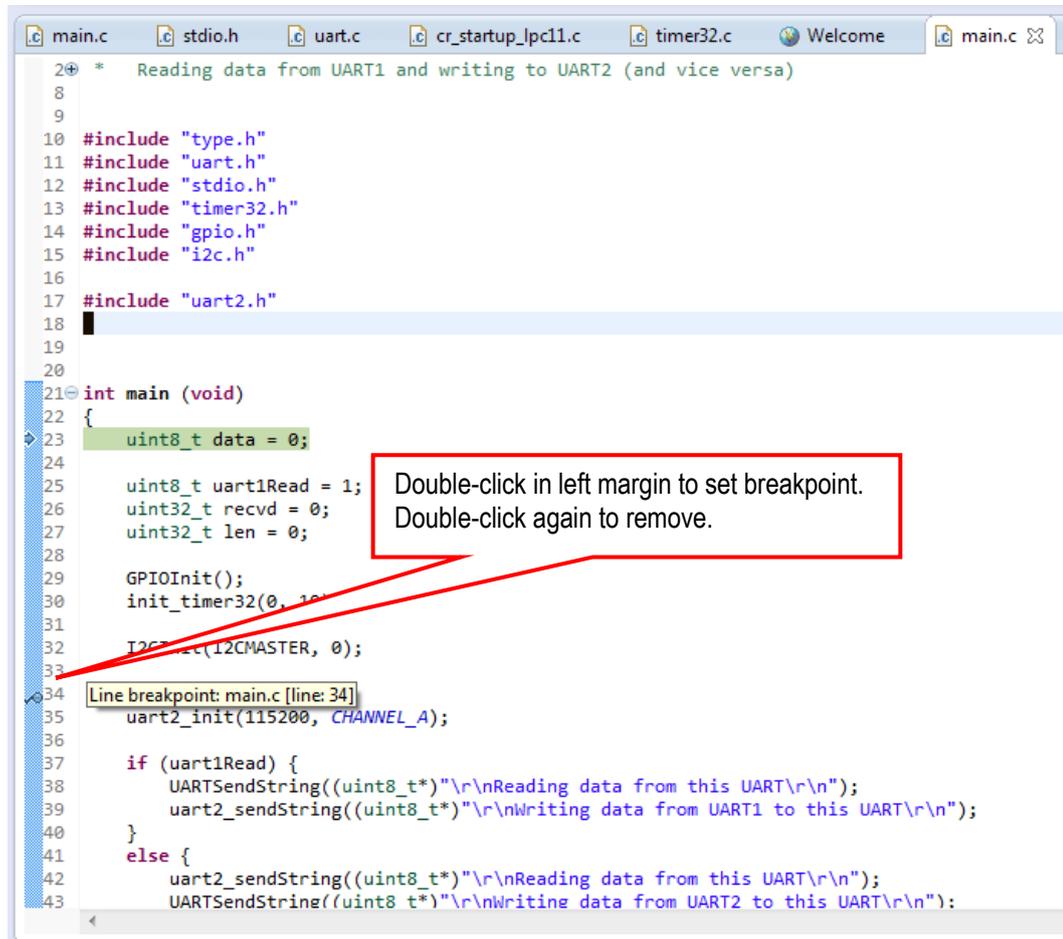


Figure 82 – LPCXpresso IDE Set Breakpoint

Pressing the **Start** button (or F8) will start execution. Hitting a breakpoint will stop execution. Figure 83 below illustrates what the call stack looks like after stopping at line #34 in main().

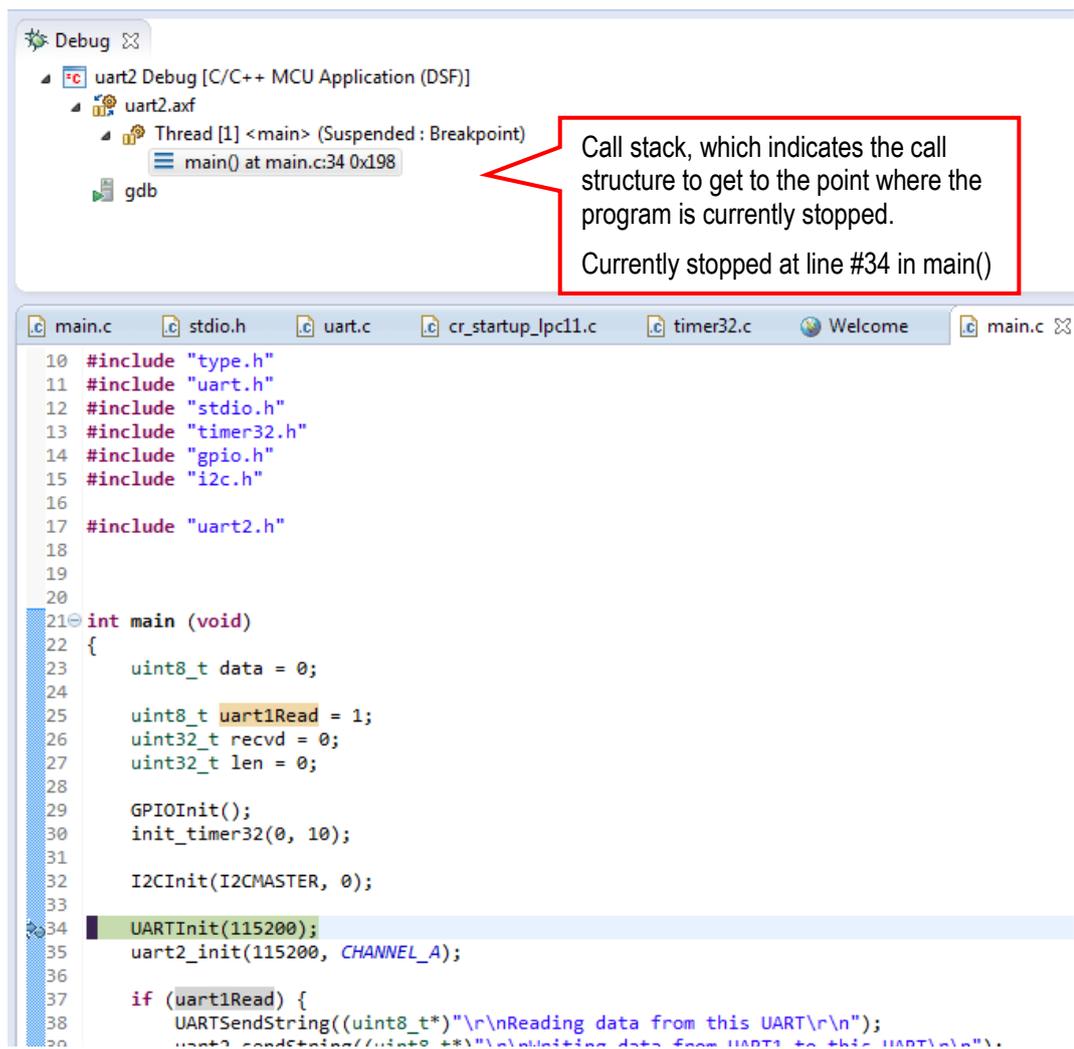


Figure 83 – LPCXpresso IDE Run to Breakpoint

Figure 84 below illustrates what the call stack can look like when the call depth is a little deeper, 6 levels in this case. It also illustrates that it is possible to hover the mouse cursor over a variable. A variable window will then pop up showing the current variable value.

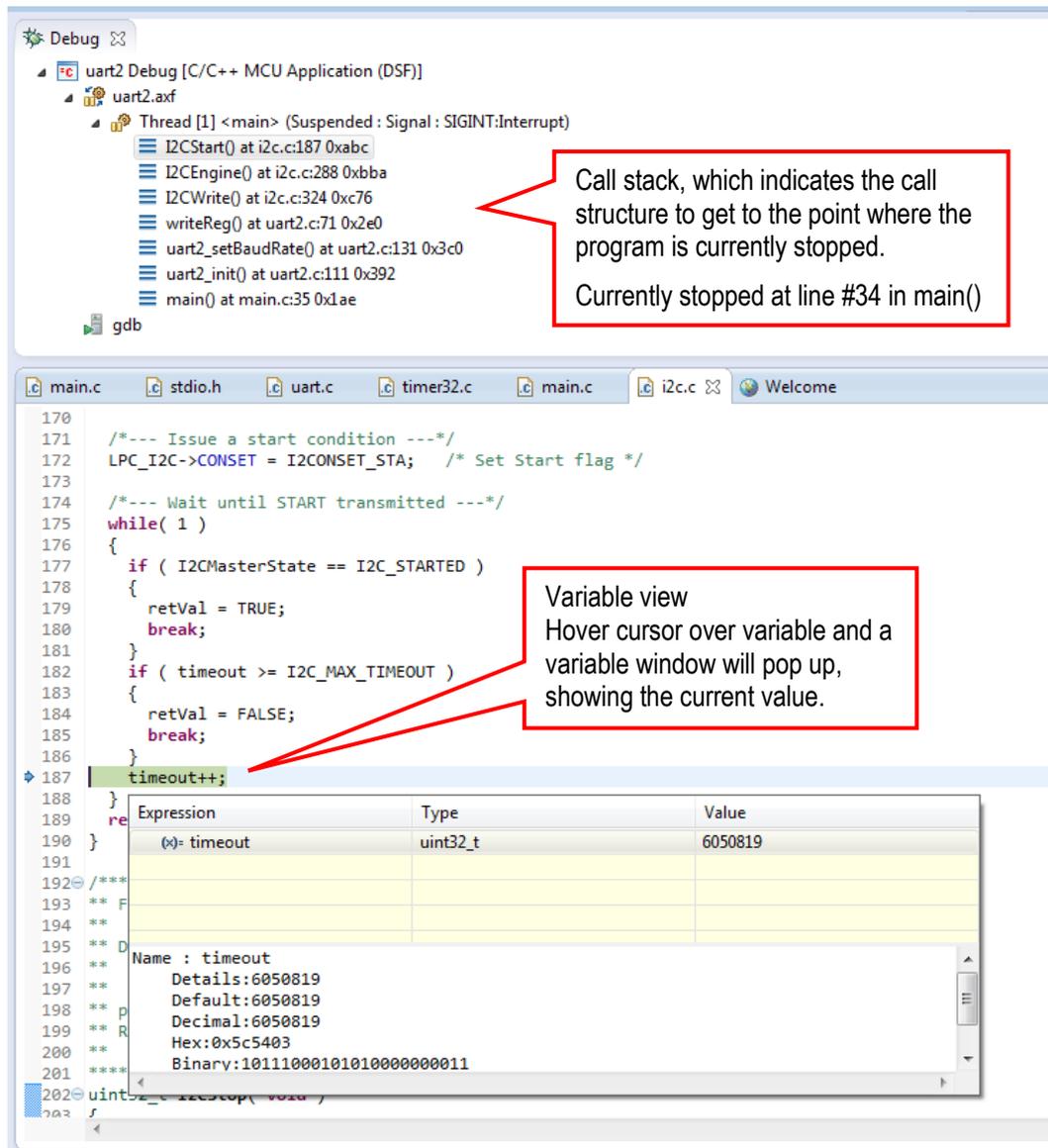


Figure 84 – LPCXpresso IDE Variable View

9.3.1 Downloading Just Code

This section describes how to download an application to the LPCXpresso board, i.e., to the LPC111x, without also starting a debug session.

Click on the "Program Flash" icon from the tool bar, see picture below. The icon can be at different places depending on window size.

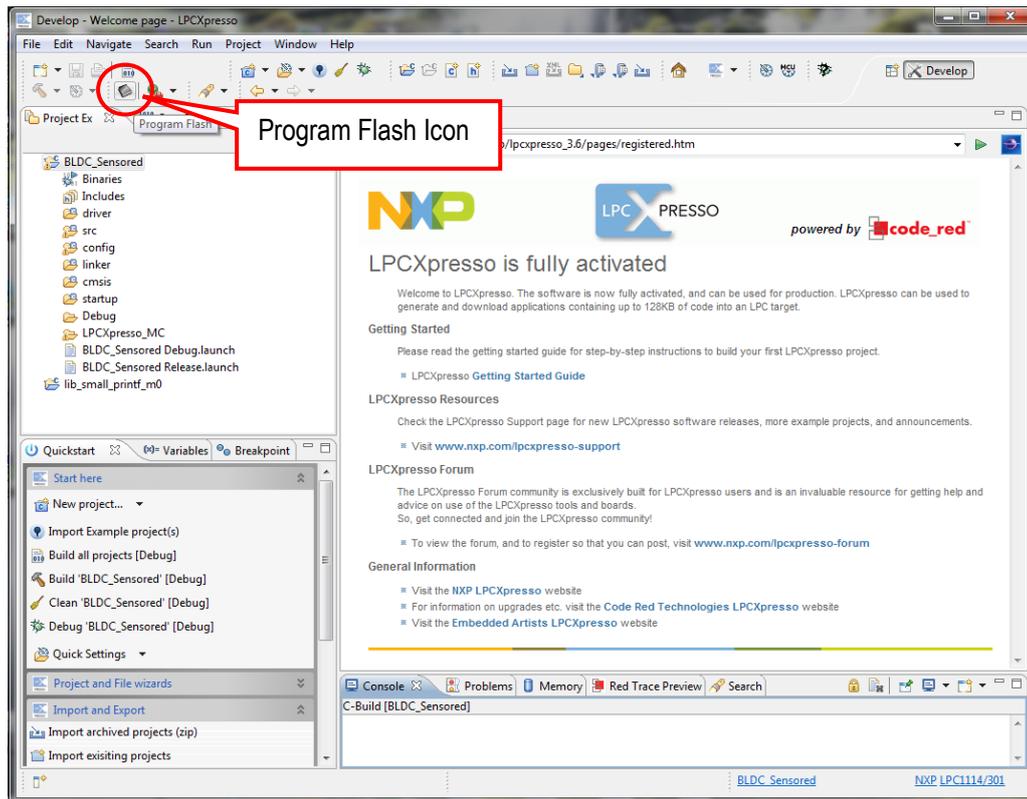


Figure 85 – LPCXpresso IDE Program Flash Icon

The next step is to select which processor to download to. Select **LPC1115** or **LPC1114** from the list that is presented. Then press OK button. Note that this step is sometimes not needed because the LPCXpresso IDE can itself detect which processor it is connected to.

The next step is to browse to the file to download. Press the “Browse” button.

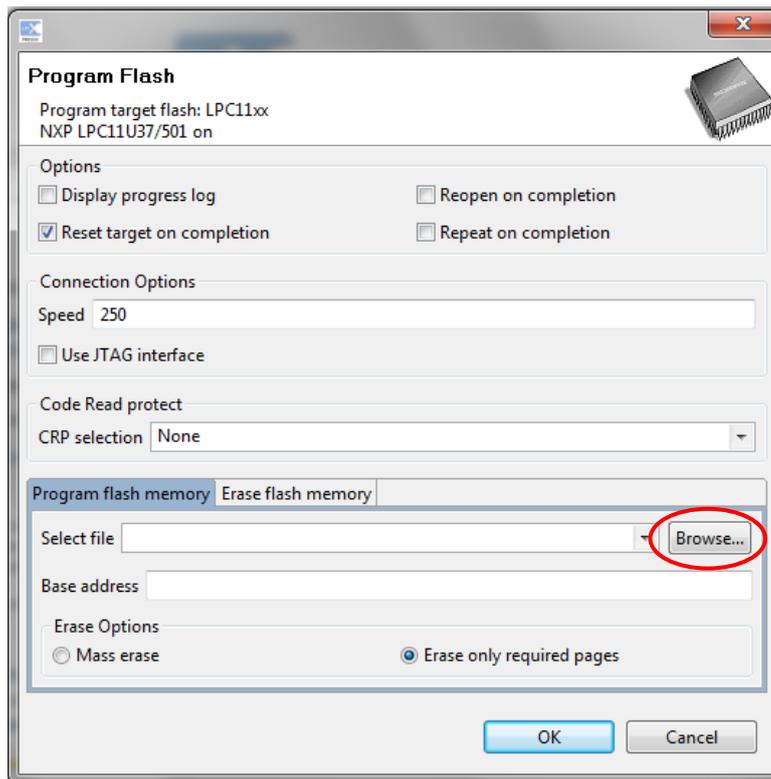


Figure 86 – LPCXpresso IDE Program Flash Window

Browse to the projects top directory and then “Debug”. In this subfolder there is either a file ending with *.axf or *.bin. Select one of these files. Press the “Open” button.

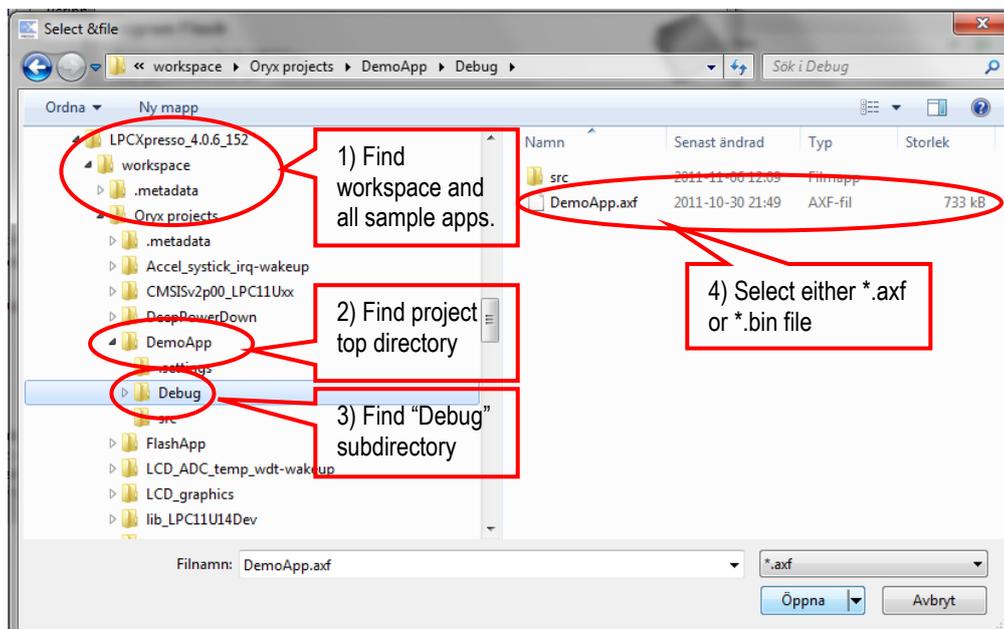


Figure 87 – Browse to File to Download

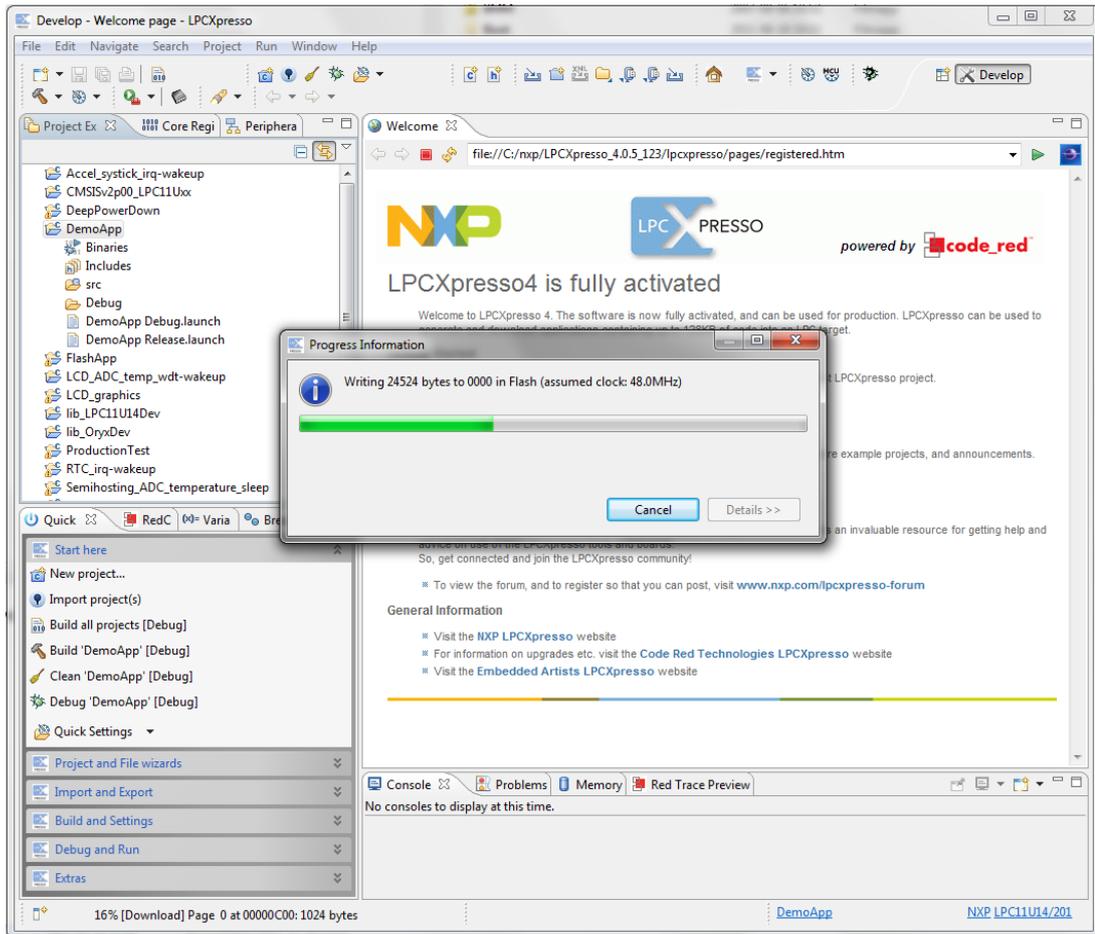


Figure 88 – LPCXpresso IDE Program Flashing in Progress

In case flashing fails, an error message like below will be displayed. This is an indication that the debugger could not connect to the LPC111x. The most common reason is that the microcontroller is in a low-power mode where the debug connection is disabled. Make sure the microcontroller is in ISP/bootload mode and try again. This is accomplished by pulling pin PIO0_1 low (via 100-1000 ohm resistor to ground).

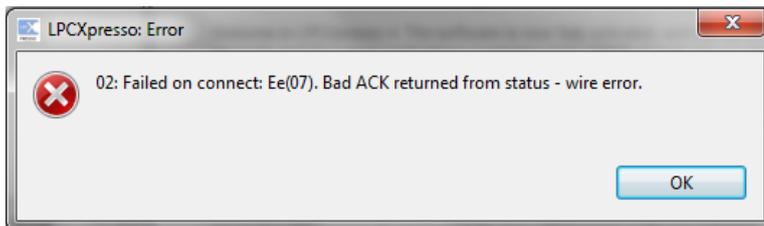


Figure 89 – LPCXpresso IDE Program Failing to Flash

There is an alternative way of initiating the program download process. From the workspace, right click on the *.axf or *.bin file (found under the “Debug” subdirectory). Then select “Binary Utility” and “Program Flash”.

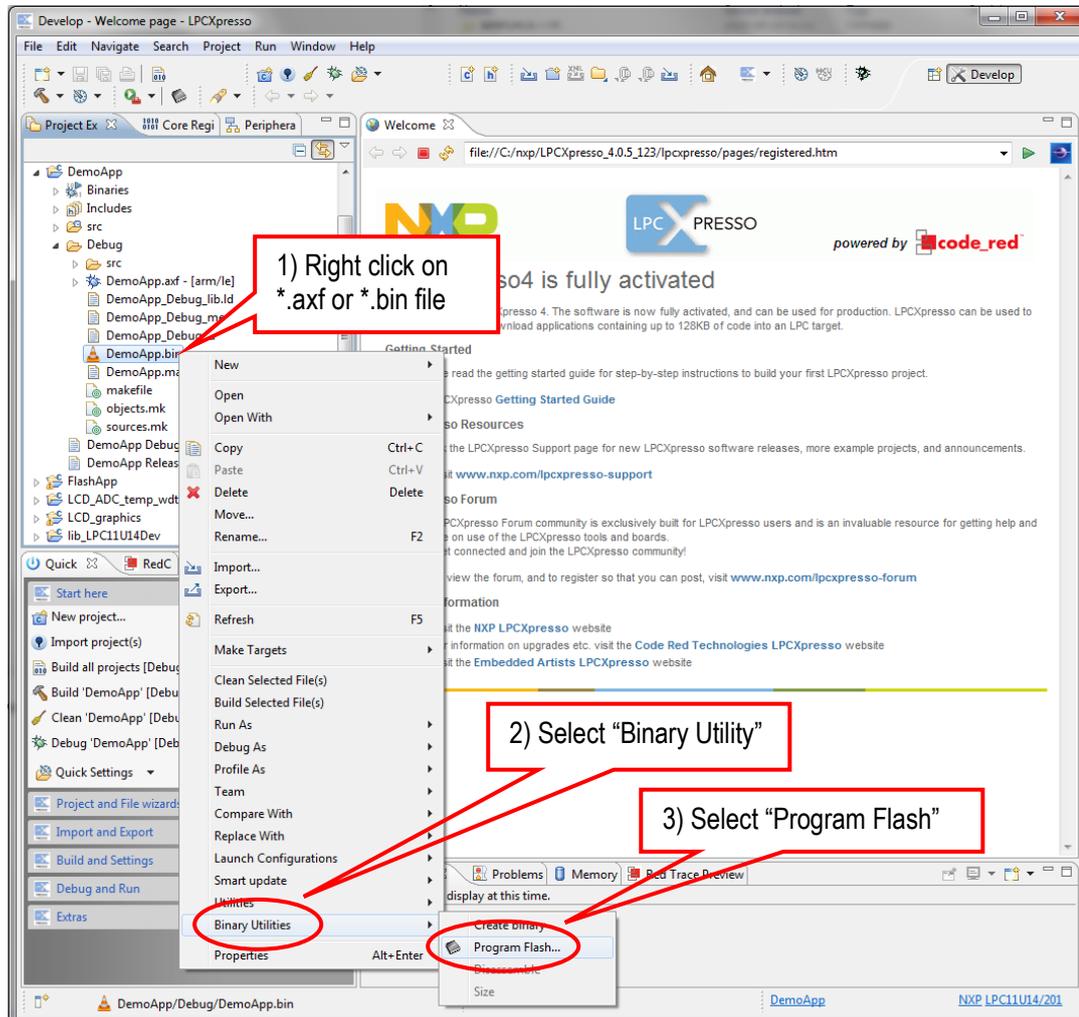


Figure 90 – LPCpresso IDE Binary Utility

9.4 Create own Projects by Copy Existing Project

The simplest way to create a new project is to copy an existing project. Start by right-clicking on a suitable existing project to start from. Select *Copy*, as illustrated in Figure 91 below. Then right-click on an empty space in the *Project Explorer* window and select *Paste*, as illustrated in Figure 92 below. Finally give the new project a suitable name, as illustrated in Figure 93 below.

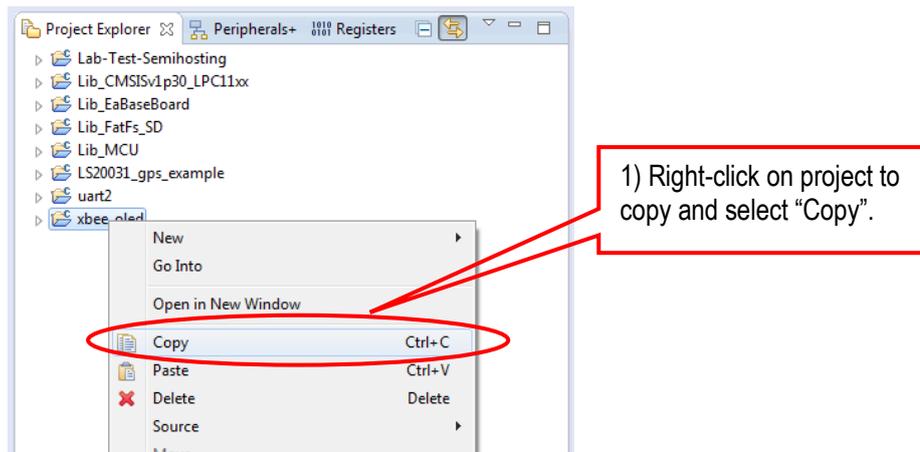


Figure 91 – Copy Existing Project

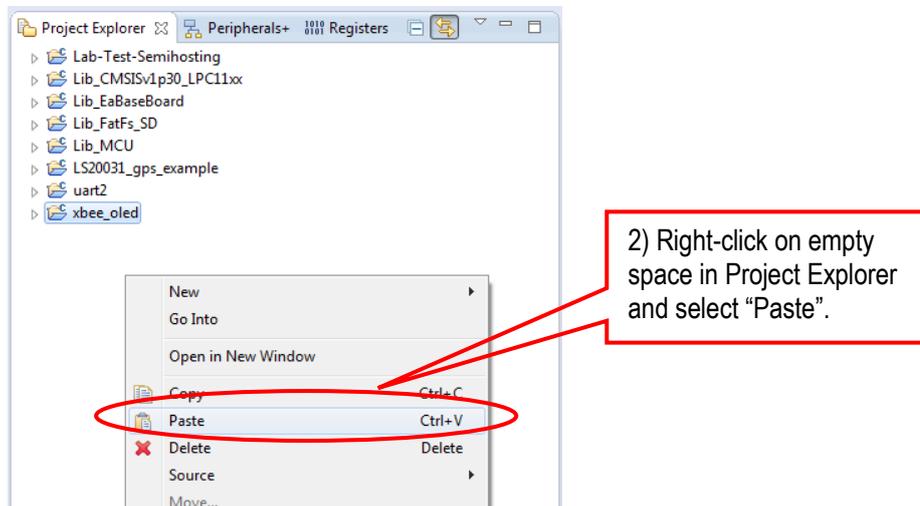


Figure 92 – Paste Project

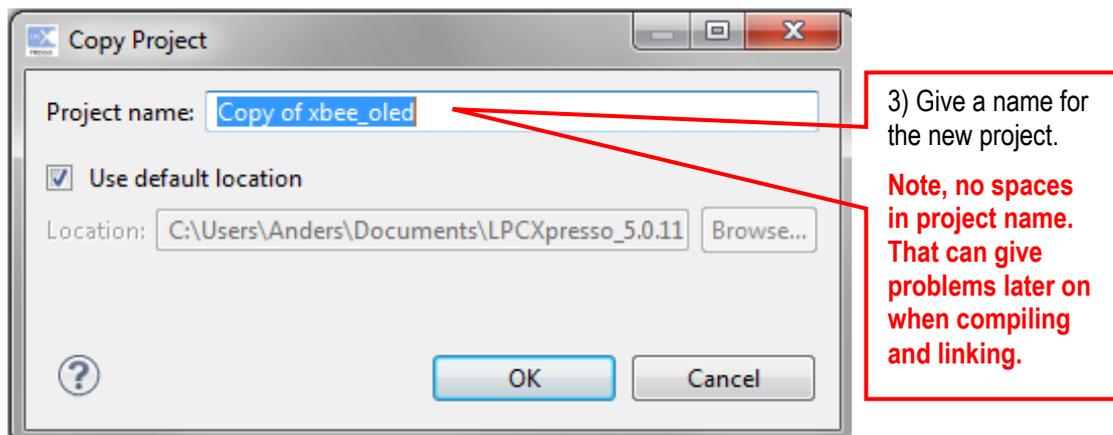


Figure 93 – Copy Project and Rename

9.5 Common Problems

In this section a number of common problems are listed.

9.5.1 Error message: Failed on chip setup

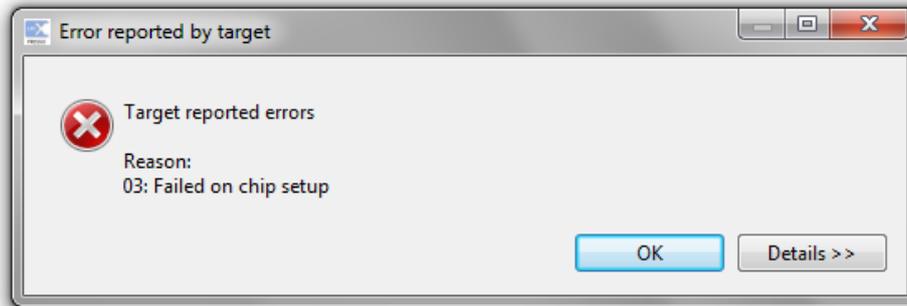


Figure 94 – LPCXpresso IDE Error: Failed on chip setup

In the console window more detailed information is given. It can for example look like this:

```
Invalid LPC1114/301 Part ID: 0x00050080  
Known LPC1114/301 ID(s): 0x0444102B, 0x2540102B  
03: Failed on chip setup: Ec(01). Invalid, mismatched, or unknown part
```

The solution is to change the chip type to the correct version. Click on the project main folder on in the Project Explorer window to the left and press Alt+Enter. Alternatively left click on the project folder and select *Properties* (at the end of the list).

Select the *C/C++ Build* menu and then the *MCU setting* menu. A list of available chips is presented. Select the correct chip in this list. Note that it is possible to select the chips that are listed in red. There are however restrictions for these chips. Typically in how big the code size can be.

10 Further Information

The LPC111x microcontroller is a complex circuit and there exist a number of other documents with a lot more information. The following documents are recommended as a complement to this document.

- [1] NXP LPC111x Information (Datasheet, User's Manual and Errata)
<http://ics.nxp.com/products/lpc1000/lpc1100/lpc11cxx/>
- [2] ARM Processor Documentation
Documentation from ARM can be found at: <http://infocenter.arm.com/>.
- [3] Information on different ARM Architectures
<http://www.arm.com/products/processors/technologies/instruction-set-architectures.php>
- [4] ARMv7-M Architecture Reference Manual. Document identity: DDI 0403D
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403c/index.html>
- [5] ARMv6-M Architecture Reference Manual. Document identity: DDI 0419B
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419b/index.html>
- [6] Cortex-M0 Technical Reference Manual. Revision: r0p0
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0432c/index.html>
- [7] LPCXpresso IDE: NXP's low-cost development platform for LPC families, which is an Eclipse-based IDE.
<http://ics.nxp.com/lpcxpresso/>
- [8] LPC1000 Yahoo Group. A discussion forum dedicated entirely to the NXP LPC1xxx series of microcontrollers.
<http://tech.groups.yahoo.com/group/lpc1000/>
- [9] LPC2000 Yahoo Group. A discussion forum dedicated entirely to the NXP LPC2xxx series of microcontrollers. This group might be more active than the LPC1000 group.
<http://tech.groups.yahoo.com/group/lpc2000/>
- [10] LPCware, NXP's community for developers
<http://www.lpcware.com/>

Note that there can be newer versions of the documents than the ones linked to here. Always check for the latest information/version.