Datalogisk Institut
KØBENHAVNS UNIVERSITET

DIKU

HogthrobV0 Users Manual

Martin Leopold

Technical Report no. 07/05
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

HogthrobV0

Users Manual

version 0.3
September 24, 2007

Martin Leopold
Department of Computer science, University of Copenhagen

Technical Report no. 07/05
ISSN: 0107-8283

Revision History		
0.1	Apr. 2006	Initial version
0.2	Oct. 2006	Pin definitions
0.3	Sep. 2007	Restructured

Contents

Chapter 1

The Hogthrob Prototype Platform

The Hogthrob prototype platform (HogthrobV0) must serve as a development platform throughout the Hogthrob project. It must be general enough to allow a large variety of configurations and robust enough to allow lab and field experiments.

The platform was defined by the Hogthrob partners and was implemented by I/O Technologies delivering practical expertise in embedded systems design, PCB[1] layout and assembly. The PCB was manufactured and assembled in a foundry before delivery. In total 50 boards are produced.

The platform was delivered in two stages. First a few boards were delivered for testing and evaluation. The testing involved developing the software to be run on the platform testing every feature of the platform. In a second stage the 50 boards are produced.

The design goals of the Hogthrob prototype platform are different from that of the sensor node we are trying to build. It must be functionally equivalent of our sensor node on a chip, and we must be able map the design to the performance of a sensor node on a chip.

The two major goal of HogthrobV0 are

- to allow software/hardware co-design
- to provide a prototype platform for further exploration of the design space.

The platform must be flexible enough to let us change any of the givens of the sensor node design: radio, sensors, microprocessor, hardware accelerators, etc. This allows us to explore a broad spectrum of design choices: hardware/software boundary, radio protocol design, duty cycling, sensor sampling frequencies, etc.

To achieve these objectives we adapt a modular design strategy so that we can swap sensors or radio transceivers with ones resulting in more efficient energy and system performance. To experiment with microprocessor designs and/or hardware accelerators, we need some form of reconfigurable logic on the prototype platform. To sum up our strategy for building a platform with no constraints:

- Configurable logic (FPGA) to develop hardware
- A/D[2] converter (rarely included in configurable logic blocks)

[1]Printed Circuitry Board
[2]Analog to digital

- A low-power timer
- Add on-board with wireless communication
- Add on-board with sensors
- The ability be battery powered

We choose to implement these goals using a Xilinx Spartan III FPGA with external FLASH for the FPGA configuration and for the program running on the FPGA. In addition we placed an ATMega 182l MCU that provides A/D as well as housekeeping for the FPGA power up/down procedure.

The following Section describes the platform design, pin connections and the general procedure to setup the platform eg. cables, power source, etc. Programming the FPGA is described in Section 3. Programming the ATMega is described in Section 4.

1.1 Further Information

Apart from this manual the following documents provide further information about the platform:

- *Martin Hansen*: System Design Specification for 1087_Hogthrob (unpublished)
- *Kashif Virk*: Testing FPGA Interfaces on Hogthrob Development Platform[4]

Chapter 2

HogthrobV0 Overview

The functionality of the platform can be divided into four closely interacting subsystems: computing, sensing, communication, and power supply (see Figure 2.1). We will look into the details of each of these subsystems in the following, fist let us sum up the contents of this division:

Computing an FPGA for hardware development and an MCU with A/D converter for external peripherals

Communication an add on-board with a flexible radio with low level access

Sensing an add on-board with sensors

Power a power supply allowing battery powered operation while maintaining a steady supply.

Figure 2.1 depicts each of these subsystems and their interconnection, Figure 2.2 depicts the layout on the board and external connections.

2.1 External Connections and Cables

HoghtrobV0 has 6 pin headers for external connections:

J1 Combined programming of ATMega and JTAG bus (FPGA, PROMs)

J2 16 general I/O pins for FPGA

J3 ATMega analog input and digital I/O

J4 15 general I/O pins for FPGA

J5 Radio power

J6 Radio I/O

J7 External 5 V power supply (see below for polarity).

2.1.1 Power

The board is powered through a common power connector (see figure 2.3(a)). The input power may not exceed 5.5 V and it is essential that polarity is not reversed (positive connected to gnd). If in doubt insert a diode in the power cable to avoid glitches.

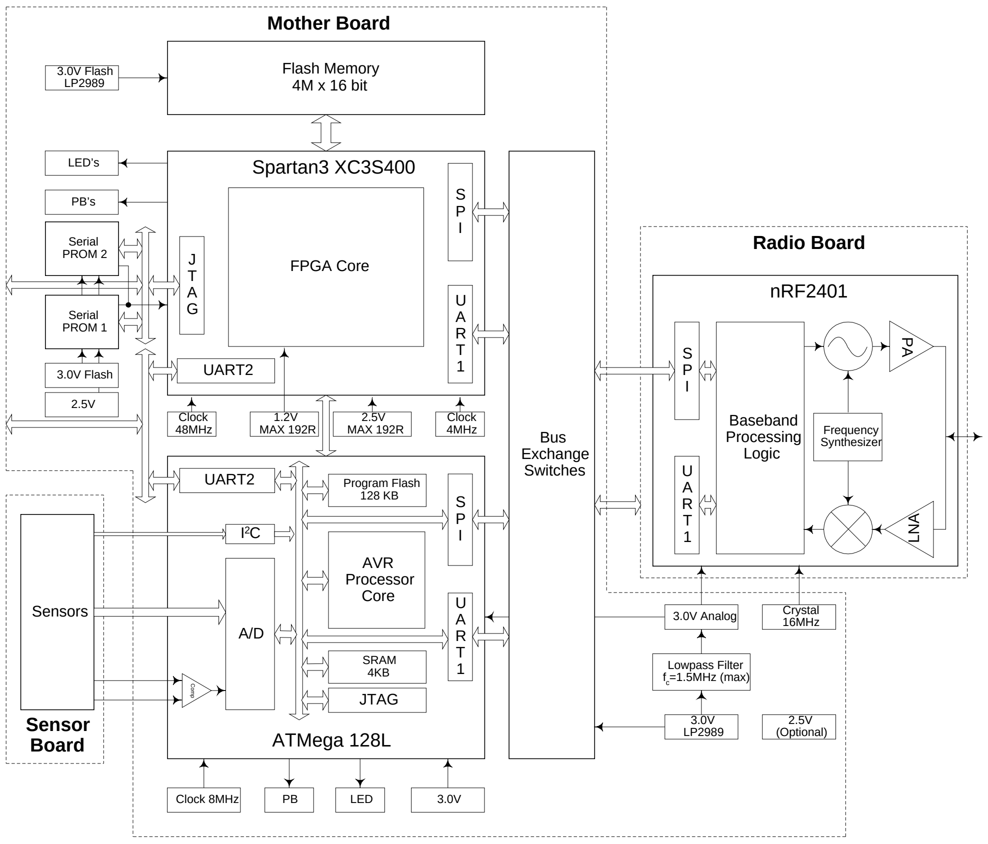
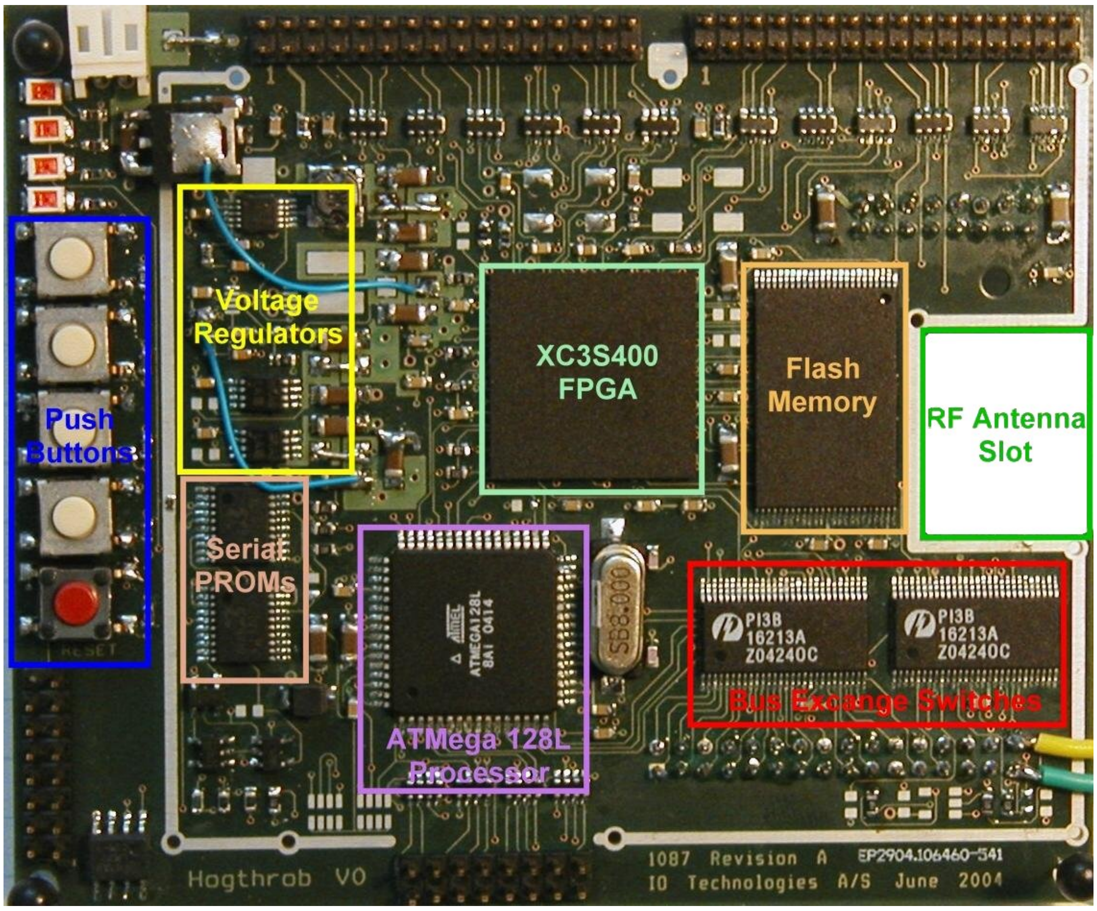
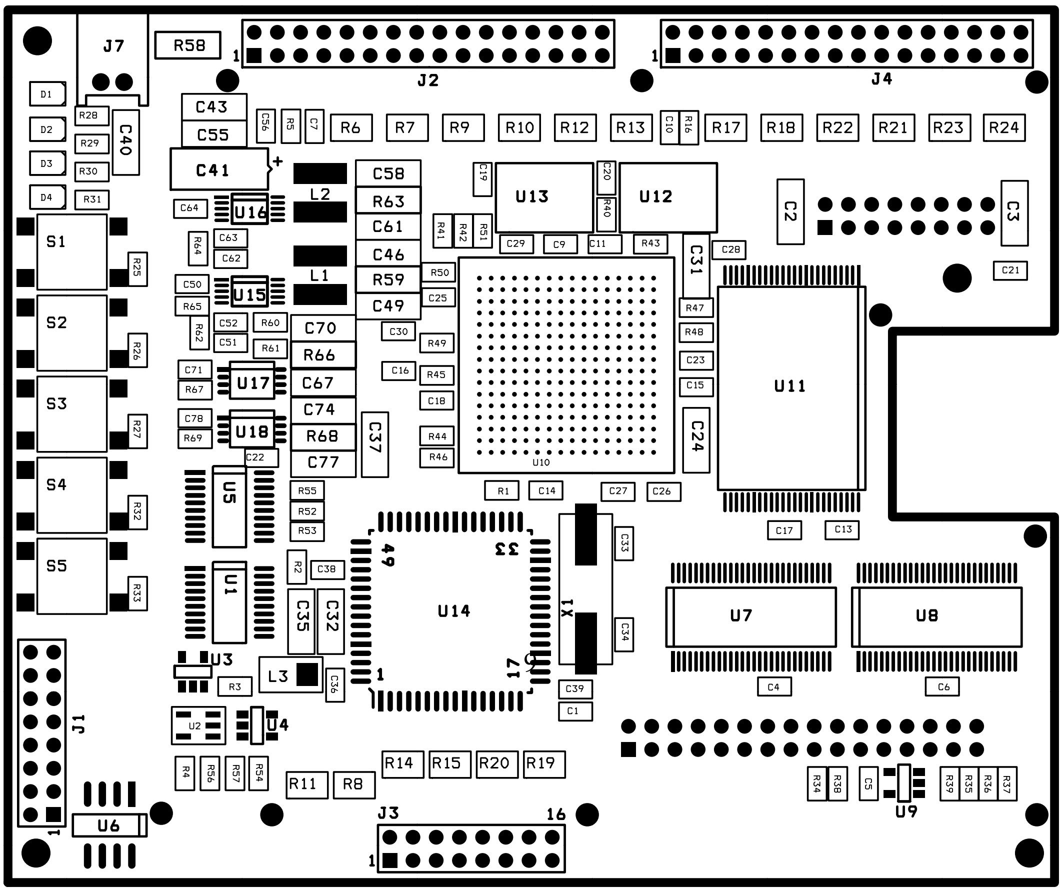


Figure 2.1 *HogthrobV0 interconnections (figure by Kashif Virk))*

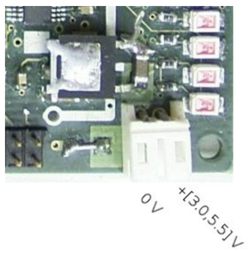


(a) Motherboard (Picture and annotation by Kashif Virk)

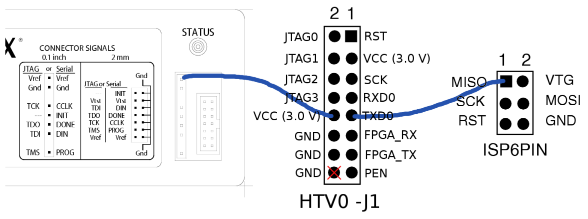


(b) Layout

Figure 2.2. – *Hogthrob prototype platform HogthrobV0*



(a) J5 power connector



(b) J1 programming connector

Figure 2.3 *Power and programming cables*

2.1.2 Programming

Connector J1 combines programming interfaces for the ATMega, FPGA and configuration PROMs (see figure 2.3(b)). Pin 16 has been cut and pin 16 has been blocked to provide a key for the correct placement of the connector. The two cables are combined into one that is connected to the appropriated programmers when programming either device.

As depicted in Figure 2.3(b) the FPGA is connected to a Xilinx[1] programming dongle, such as *Platform Cable USB* or *Parallel Cable IV*. The ATMega is connected to an Atmel[2] programming dongle such as the STK-500 or AVR-ISP. connected to

JTAG	HTV0 J1	STK-500 (ISP)
	1: RESET	RST
TDO	2: FPGA_TDO (JTAG0)	
	3: 3.0 V	
TDI	4: TDI (JTAG1)	
	5: SCK	SCK
TMS	6: TMS (JTAG2)	
	7: RXD0/DPI	MOSI
TCK	8: TCK (JTAG3)	
	9: TXD0/DPO	MISO
V_ref	10: 3.0 V	
	11: FPGA_RX	
Gnd	12: GND	
	13: FPGA_TX	
	14: GND	GND
	15: PEN	(only for SPI protramming)
	16: GND (Pin cut)	

2.1.3 UART

The ATMega and FPGA have a UART connection to the J1 pin header. The voltage levels of these connections are different to those of a PC-RS232 UART and requires level conversion to

[1]http://www.xilinx.com
[2]http://www.atmel.com

be connected to a PC.

Some programmer boards such as the Atmel STK-500[3] feature built in level conversion, however using a USB-RS232 converter can be convenient. Such a converter is usually built using a general purpose serial conversion chip such as Prolific PL-2303[4] or FTDI FT-232[5], the inputs on the chip can be connected to the UART output of the platform.

2.2 Pin Connections

End point	AVR	Line
J1	PD0	SCL
J1	PD1	SDA
J1	PD2	RXD1
J1	PD3	RXD1
LED	PD7	LED
Button	PE7	BOTTON
	PB4	DONE (FPGA booted)

2.2.1 ATMega-FPGA

Line	AVR	FPGA
Power_on	PD6	High means FPGA on
DONE	PB4	B14 (VCCAUX_DONE)
FPGA_CS	PC7	P7 (B5_IO)
FPGA_INT6	PE6	N5 (B5_IO)
ALE	PG2	T5 (B5_IO
RD_N (PROM_SEL)	PG1	T4 (B5_IO_L01P_5/CS_B)
WE_N	PG0	T3 (B5_IO_L01N_5/RDWR_B)
DA0-DA7	PA0-PA7	P5, N6, M6, B6, N7, M7, T7, B7

2.2.2 ATMega-Radio

RADIO_IO11 used to select RX/TX direction if RXD1 and TXD1 is used as a combined single-wire UART.

[3]http://www.atmel.com
[4]http://www.prolific.com.tw
[5]http://www.ftdichip.com

Line	Radio Board	AVR	FPGA
RXD1	J1, pin 31	PD2 (RXD1/INT2)	FPGA_IO17
TXD1	J1, pin 32	PD3 (TXD1/INT3)	FPGA_IO18
CLK (SCK)	CLK1	PB1 (CLK)	FPGA_IO13
MOSI+MISO	DATA	PB2+PB3	FPGA_IO14+FPGA_IO15
INT0	DR2	PE4 (INT4)	FPGA_IO1
Radio_IO0	DR1	PE5 (INT5) + (by mistake) PB0 ($\hat{S}$S)	FPGA_IO4
Radio_IO1	DOUT2	PB5	FPGA_IO2
Radio_IO2	CS	PB6	FPGA_IO3
Radio_IO3	PWR_UP	PB7	FPGA_IO5
	J1, pin 13		FPGA_IO6
Radio_IO4	J1, pin 15	PD4	FPGA_IO7
Radio_IO5	DEVICE_CODE0	PD5	FPGA_IO8
Radio_IO6	DEVICE_CODE1	PG3	FPGA_IO9
Radio_IO7	LED0	PG4	FPGA_IO10
Radio_IO8	LED1	PC3	FPGA_IO11
Radio_IO9	CLK2	PC2	FPGA_IO12
Radio_IO10	CE	PC1	FPGA_IO0
Radio_IO11	NC7SZ125	PC0	FPGA_IO16

2.2.3 Bus Switches

	RADIO_MUXS2	**RADIO_MUXS1**	**RADIO_MUXS0**
PI3B16213	S2	S1	S0
RADIO_MUXx	2	1	0
AVR - pin	PC6	PC5	PC4

2.2.4 FPGA IO

Line	Bank	Pin	Connector	Line	Bank	Pin	Connector
FPGA_AIO0	B6	K1	J2, pin 5	FPGA_BIO	B7	G2	J4, pin 5
FPGA_AIO1	B6	R1	J2, pin 7	FPGA_BIO1	B7	C1	J4, pin 7
FPGA_AIO2	B6	P1	J2, pin 9	FPGA_BIO2	B7	R1	J4, pin 9
FPGA_AIO3	B6	P2	J2, pin 11	FPGA_BIO3	B7	C2	J4, pin 11
FPGA_AIO4	B6	N3	J2, pin 15	FPGA_BIO4	B7	C3	J4, pin 15
FPGA_AIO5	B6	N2	J2, pin 17	FPGA_BIO5	B7	D1	J4, pin 17
FPGA_AIO6	B6	N1	J2, pin 19	FPGA_BIO6	B7	D2	J4, pin 19
FPGA_AIO7	B6	M4	J2, pin 21	FPGA_BIO7	B7	F3	J4, pin 21
FPGA_AIO8	B6	M3	J2, pin 25	FPGA_BIO8	B7	D3	J4, pin 25
FPGA_AIO9	B6	M2	J2, pin 27	FPGA_BIO9	B7	F1	J4, pin 27
FPGA_AIO10	B6	M1	J2, pin 29	FPGA_BIO10	B7	F2	J4, pin 29
FPGA_AIO11	B6	L5	J2, pin 31	FPGA_BIO11	B7	F4	J4, pin 31
FPGA_AIO12	B6	L4	J2, pin 32	FPGA_BIO12	B7	F4	J4, pin 32
FPGA_AIO13	B6	L3	J2, pin 30	FPGA_BIO13	B7	F2	J4, pin 30
FPGA_AIO14	B6	L2	J2, pin 28	FPGA_BIO14	B7	F3	J4, pin 28
FPGA_AIO15	B6	K6	J2, pin 26	FPGA_BIO15	B7	G5	J4, pin 26
FPGA_AIO16	B6	K4	J2, pin 22	FPGA_BIO16	B7	F5	J4, pin 22
FPGA_AIO17	B6	K3	J2, pin 20	FPGA_BIO17	B7	G3	J4, pin 20
FPGA_AIO18	B6	K2	J2, pin 18	FPGA_BIO18	B7	G4	J4, pin 18
FPGA_AIO19	B6	J4	J2, pin 16	FPGA_BIO19	B7	H3	J4, pin 16
FPGA_AIO20	B6	J3	J2, pin 12	FPGA_BIO20	B7	H4	J4, pin 12
FPGA_AIO21	B6	J2	J2, pin 10	FPGA_AIO21	B7	H1	J4, pin 10

Chapter 3

Xilinx FPGA

The FPGA portion of the platform is controlled by the ATMega in a number of ways. The ATMega powers the FPGA on and off and points the radio interface to either the ATMega of FPGA. The FPGA boot procedure and ATMega dependence is described in Section 4.2 on page 20, the following describes the features that are independent of the ATMega.

In order to program the FPGA the appropriate compiler must be installed see Section 3.2.

3.1 Hardware Setup

Once the FPGA has been booted it operates independently of the ATMega. It features a large number of digital I/O lines, buttons, leds, external UART connection and is connected to an external FLASH. See schematics in Appendix A.

3.1.1 Configuration Flash

On boot the FPGA loads a configuration from the configuration FLASH. The board has one flash chip mounted and solder pads for an optional additional FLASH chip. Chip selection is performed by the ATMega (described in Section 4) and is required before booting the FPGA.

The FLASH chip is programmed using Xilinx Impact using the JTAG connector.

3.1.2 Program Flash and ATMega interface

The FPGA is connected to the program flash and ATMega using elaborate interfaces. Using these interfaces requires including an appropriate controller in the FPGA logic, that we will not cover here. The detailed FPGA testing procedure**??** describes an example of such controllers.

The FGPA is connected to the ATMega by a set of pins that can be treated as either general purpose I/O pins or as external memory to the ATMega. This interface uses a set of pins to alternate between address and data, this is controlled automatically by the ATMega, but must be programmed accordingly on the FPGA.

Programs for the CPU running on the FPGA can be stored either in block-memory or via the external FLASH. This requires that the FPGA implements a FLASH controller for storing an retrieving data from the FLASH.

3.1.3 Clock Source

The FPGA has two external crystal clock sources: a 4 MHz and a 48 MHz source. The 4 MHz clock is always enabled, but the 48 MHz must be enabled by pulling I/O pin R10 high.

3.1.4 FPGA I/O

A large number of the FPGA I/O pins have been connected to external connectors either with a dedicated purpose of as general purpose I/O. An example constrains file with appropriate naming for the external I/O pins is given in Appendix D on page 50, as well as [4].

Each of the FPGA I/O blocks must be supplied with an external power source. The blocks B0, B1, B2, B3, B4 and B5 (that are used internally on the board) are all supplied on the board while B6 and B7 must be feed through the connectors J2 and J4 respectively using a jumper.

Block	Power	Connection
B0	Internal (3.0 V)	FLASH
B1	Internal (3.0 V)	FLASH
B2	Internal (3.0 V)	FLASH + FPGA UART (connector J1)
B3	Internal (3.0 V)	Radio via J6
B4	Internal (2.5 V)	Buttons + LEDs
B5	Internal (3.0 V)	ATMega
B6	External through pin 3 of J2	J2
B7	External through pin 3 of J4	J4
B8	Grounded	

Buttons and Leds

The FPGA is connected to 3 buttons and 3 leds, the LEDs are active high, while the buttons are low when pressed. Please note this is a different semantics than other buttons and leds found on the board. See Chapter 4.

FPGA UART

The FPGA has two pins connected to the common programming connector J1 intended for use as a UART. The two pins are F14 and F15, these pins are part of the I/O block B2 resulting in an 3.0 V level on these two pins.

3.2 Software installation

To program the FPGA the ISE tool-suit from Xilinx is required. A full version supporting all Xilinx devices or a limited "WebPack" version only supporting smaller devices can be downloaded. The WebPack should be sufficient for our device.

ISE WebPack is free of charge, but you are required to register with Xilinx in order to download it. At the time of writing the latest version (8.1i) is available for Windows, Linux and Solaris. The Windows is self explanatory, while the Linux installtion can be a bit tricky.

3.2.1 ISE 8.1 for Linux

Use the provided installer (either web-install or full download), this will provide you with a `Xilinx` directory containing the tools. The installer also compiles and installs drivers, which requires you to run the installer as root.

The installer is provided for Red Hat Linux, but it should work perfectly on most other distributions, except for the driver setup scripts. The installer contains scripts to setup the drivers at boot time and these are unlikely to be setup correctly outside of Red Hat.

The installation script will also prepare an environment settings that you will need to load to start the tools. Depending on your shell this will look something like this:

```
 source ~/Xilinx/setting.sh
```

USB Cable Drivers

The drivers provided by Xilinx at the time of writing only support Linux kernel version prior to 2.6.13. For more recent kernel versions you need to obtain one driver from from Jungo.com directly and one from the Xilinx distribution.

For recent kernel versions get the "driver development kit" and compile this. For gcc versions 4.0 and up find the macro "KBUILD_STR" in the Makefile and remove it.

```
tar zxfv WD802ln.tgz
cd WinDriver/redist
./configure --with-kernel-source=/usr/src/linux-headers-2.6.15-26-386
make
```

For udev enabled platforms a udev rule must be created for the driver to be loaded. Create a udev rule in `/etc/udev/rules.d` for example `10-xilinx.rules`

```
BUS=="usb", SYSFS{idVendor}=="03fd", SYSFS{idProduct}=="0007",\\
  RUN+="/sbin/fxload -v -t fx2 -I /usr/local/Xilinx/bin/lin/xusbdfwu.hex
```

Parallel Cable Drivers

The drivers for the parallel and USB cables are installed as part of the installation procedure. If this should fail or if you upgrade your kernel you can download the drivers from one of the following locations depending on your kernel version. Unpack theses archives and build the drivers with make:

```
ftp://ftp.xilinx.com/pub/utilities/fpga/linuxdrivers.tar.gz
```

For Linux 2.6

```
ftp://ftp.xilinx.com/pub/utilities/M1_workstation/linuxdrivers.2.6.tar.gz
```

The driver installation creates two scripts to load the drivers and setup the required device nodes. However the permissions of these device nodes are set such that only root can access the cables.

```
/lib/modules/misc/install_windrvr6 windrvr6
/lib/modules/misc/install_xpc4drvr
chmod 777 /dev/xpc4* /dev/windrvr6
```

Pace

Pace (the constraints editor) as of version 8.1i unfortunately suffers from a few quirks that need to be taken into account. First of all make sure that the library `libmotif3` is install, secondly make sure to setup your DISPLAY DISPLAY=:0

3.2.2 ModelSim

In addition to ISE it might be useful to simulate a project for debugging. For this purpose Xilinx provides ModelSim Xilinx Edition-III free of charge[1].

Don't forget to select the free "Starter" version when installing. And be sure to select VHDL and not Verilog. The license file is then requested with a web link in the start menu.

3.3 Building a Project

Building the project can be built using the Xilinx ISE graphical environment or it can be built using the corresponding command line tools. Either way the two methods go through the same steps. An example make file has been provided as Appendix E.

Along with the platform two example projects have been provided: An 8051 core *Oregano* and an AVR core *Nimbus*

3.3.1 Oregano

Oregano is an open source, freely available 8051 core[2] adapted to the Xilinx environment. It is a straight forward 8051 implementation with a few external peripherals such as UART.

Boot Loading Programs

Provided with the Oregano 8051 is a boot loader example design (RAMLOAD) this program accepts programs from the UART and stores them in the code memory space of the 8051. This boot loader resides in the internal memory (ROM) of the 8051 and is booted as the first program.

At start-up the boot loader emits a "=" and expects a program to be uploaded once the upload is complete it emits a ":" and waits for a command to start executing further[1].

Send a "/2000" to start the program at address 2000

If the address is accepted it sends a "@" before jumping to the address

Uploading Programs

Building programs for oregano is the same as for any other 8051 core. Compile the program using an 8051 compiler such as Keil PK51[3] and upload the program to the FPGA board using the UART:

```
cat hathat.hex > /dev/ttyUSB0
echo -en "/2000\r" > /dev/ttyUSB0
```

[1]http://www.xilinx.com/ise/optional_prod/mxe.htm
[2]http://www.oregano.at/
[3]http://www.keil.com

Chapter 4

ATMega

In this section we will concern our selves with programming the features of the HTV0 board dealing with the ATMega. We will present programs them in C (see Appendix C) and construct TinyOS interface to control the features (see Section 4.4).

4.1 Hardware Setup

The ATMega is powered on and boots a program from FLASH as soon as the board is powered up. It is then up to the ATMega program to turn on a, power up the FPGA and so forth.

The ATMega execution is controlled by a set of *fuses* before booting the first program these fuses must be set to match the configuration of the chip.

4.1.1 Fuses

The internal fuses of the ATMega change certain properties of the ATMega: available clock sources, external interfaces (ISP, JTAG, etc.), and more. These settings are pre-programmed and cannot be altered by the ATMega. Besides controlling the functionality of the ATMega, the fuse settings can affect power consumption. In our case the FPGA is by far going to be the dominating factor, making this aspect less important. Consider for example the JTAG interface (on port-F) - this peripheral could be disabled to make sure that is does not consume any power, on the other hand we are not sure how this platform is going to be used, so disabling JTAG might be a problem.

A set of fuse settings is provided in Table 4.1 that enables most features, sets up clock sources, brownout detection, etc. The setting are programmed using AVR Studio for Windows or using uisp:

```
uisp -dserial=/dev/ttyUSB0 -dpart=ATmega128 -dprog=stk500
    --wr_fuse_l=0x8e --wr_fuse_h=0x00 --wr_fuse_e=0xff
```

4.1.2 ATMega I/O

In addition to the control pins for the FPGA, the A/D pins of the ATMega are connected to the J3 pin header. Furthermore the ATMega is connected to a LED and a button.

See schematic in Appendix A for further information.

Extended fuse	0xFF	M103C OFF, WDTON OFF
High fuse	0x00	OCDEN ON, JTAGEN ON, SPIEN ON, CKOPT ON, EESAVE ON, BOOTSZ1 ON, BOOTSZ0 ON, BOOTRST OFF
Low fuse	0x8E	BODLEVEL OFF, BODEN ON, SUT1 ON, SUT0 ON CKSEL3 OFF, CKSEL2 OFF, CKSEL1 OFF, CKSEL0 ON

Table 4.1 *ATMega fuse settings (see [2, p.289] for further). The semantics of the fuse settings are reversed, meaning that a logical 1 corresponds to "off" and 0 to "on".*

Location	Component	AVR pin	Active
Radio	D1	PC3	Low
Radio	D2	PG4	Low
Motherboard	D1	PD7	High
Motherboard	B4	PE7	High = pressed

Table 4.2 *LEDs on the motherboard and radioboard*

Button and LED Interface

The platform provides 3 LEDs connected to the ATMega: one on the motherboard and two on the radio-board. The semantics of the LEDs on the radio and motherboard are different (see Table 4.2). In addition the B4 button is connected to the external interrupt 7.

External Interfaces

Analog, UART1/UART2, ISP

4.1.3 Radio Control

Selecting whether the FPGA or the ATMega uses the radio is done by two bus switches. These connect the radio either to the ATMega or to the FPGA. The two switches act as a cross-bar switch interconnecting either of 4 buses (see Figure 4.1):

A1 reset-value
A2 the radio (RF)
B1 FPGA
B2 ATMega

The relevant states of the MUX interconnect the buses as follows:

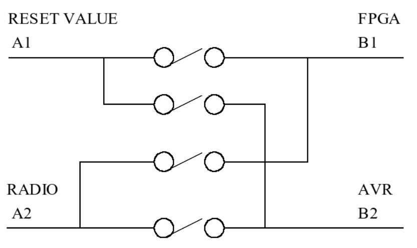


Figure 4.1 *Bus switches (Figure by Martin Hansen)*

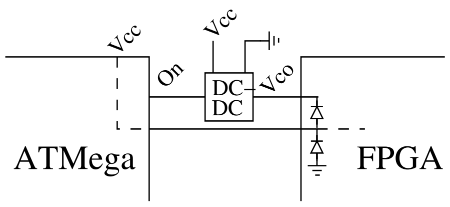


Figure 4.2 *FPGA to ATMega interconnect. Each I/O pin of the FPGA is connected with two diodes for Electro-Static Discharge protection[5].*

RF-AVR (1):	A2-B2, A1-B1 RF(A2) to AVR (B2), Reset(A1) to FPGA(B1)
RF-FPGA (2):	A2-B1, A1-B2 RF(A2) to FPGA(B1), Reset(A1) to AVR(B2)

The bus switches are controlled using 3 lines (RADIO_MUX0/1/2) connected to AVR pins PC4/PC5/PC6. The two states are selected are selected as follows:

AVR pin **RADIO_MUX_Sx**	**PC6** **2**	**PC5** **1**	**PC4** **0**
RF-AVR (1):	1	1	0
RF-FPGA (2):	1	1	1

4.2 FPGA Interface

The ATMega and FPGA are interconnected with the external memory interface of the ATMega and a few control signals. The external memory interface must be configured and enabled from the ATMega in order to be functional (see below).

As described earlier the ATMega is responsible for booting and selecting a configuration ROM for the FPGA. Let's go over the steps to boot the FPGA.

4.2.1 FPGA Power Control

The power supply to the FPGA is controlled by the ATMega. This means that in order to be functional it must first be powered up. Furthermore great care must be employed regarding the state of the interconnect-signals when the FPGA is powered off in order to prevent the FPGA from shorting the pins of the FPGA to ATMega interface to ground.

The internal circuitry of the FPGA requires that all pins connecting the FPGA and ATMega must be *tri-stated* (configured as *input*) when powering the FPGA off (see Figure 4.2).

When the `On` signal is disabled the supply voltage of the FPGA is connected to ground. If any of the pins of the ATMega are *driving* the line (that is configured as *output*) a direct connection from supply to ground has been created. This connection will draw a current and in worst case overloading and destroying the FPGA. It is thus essential that the pins of the ATMega is put into a state where they are unable to drive a current over the line (tri-stated).

4.2.2 FPGA Control Lines

The FPGA control is performed by the following lines, in short the FPGA has to be supplied with power and a prom in order to boot. The prom selection logic is probably broken and even

```
MCUCR &= ~_BV(SRW10); // No wait-states
XMCRA = 0; // No wait-states
XMCRB = _BV(XMBK) | _BV(XMM0) | _BV(XMM1) |_BV(XMM2);
// 32 kB address space w. bus-keeper
MCUCR |= _BV(SRE); // Enable external mem
```

Figure 4.3 *Configure and enable external memory*

if two proms are mounted it will probably not be possible to select the 2nd prom. In any case the line will be controlled by a pull-up resistor and leaving it will select the available PROM. In the following we will describe how to turn the FPGA on and off see Appendix C for examples in C.

In addition to the control lines below the interface also contain the data and address (DA) lines:

ALE, RD_N, WE_N External memory interface of the ATMega. This must be enabled to be functional (see above).

PROM_SEL RD_N (on the ATMega) doubles as PROM_SEL which selects which PROM to use during start-up of the FPGA. This signal should be pulled low to enable the PROM that is mounted on all boards. Even if the other PROM is mounted it might still not be possible to select it (see Appendix B).

FPGA_CS is general purpose I/O pin despite the name.

FPGA_DONE reports the successful start-up and loading of a configuration from a PROM. If the PROMS are empty the DONE signal will *not* be generated. Meaning that the ATMega cannot wait for the DONE signal if the PROM is empty!

power_on Powers up the FPGA by turning on the voltage converters for the FPGA.

FPGA Power On Procedure

Booting the FPGA is done by: (optionally) pulling PROM_SEL high, i) pulling power_on high. This causes the FPGA to turn on and load a configuration from the selected PROM. If this is successful the FPGA will set FPGA_DONE high.

FPGA Power On Procedure

Turning off the FPGA is done by first setting the interface pins to input and then powering off the device. If the external memory interface is enabled the pin direction is controlled by this unit and external memory must be disabled.

To power off: i) disable external memory ii) set all lines as input iii) set power_on low.

Enabling External Memory

The external memory of the ATMega must be configured and enabled for this peripheral unit within the ATMega to be operational. In addition the FPGA must contain a corresponding logic block. See Figure 4.3) for a C example.

```
typedef struct \{
  unsigned int      rx_en   : 1; // RX or TX operation
  unsigned int      rf_ch   : 7; // Channel frequency
  unsigned int      rf_pwr  : 2; // RF output power
  unsigned int      xo_f    : 3; // Crystal frequency
  unsigned int      rfdr_sb : 1; // RF data rate (1Mbps requires 16 MHz crystal)
  unsigned int      cm      : 1; // Communicaton mode (Direct or ShockBurst)
  unsigned int      rx2_en  : 1; // Enable two channel receive mode
  //high order bits
\} __attribute__ ((packed))      gen_config_t;
typedef struct
  gen_config_t general_config;

  unsigned int      crc_en:1;  // Enable on-chip CRC generation/checking
  unsigned int      crc_l:1;   // 8 or 16 bit CRC
  unsigned int      addr_w:6;  // Number of address bits (both RX channels)
  uint8_t           addr1[5];  // Up to 5 byte address for RX channel 1
  uint8_t           addr2[5];  // Up to 5 byte address for RX channel 2
  uint8_t           data1_w;   // Length of data payload RX channel 1 (bits)
  uint8_t           data2_w;   // Length of data payload RX channel 2 (bits)
  //uint8_t           test[3];   // reserved for testing - no need to send
  // High order bits
 __attribute__ ((packed))      shock_conf_t;
```

Figure 4.4 *Configuration words for nRF2401. The upper part show the general configuration word the lower shows the additional options for shockburst mode.*

4.3 Radio Interface

The radio is connected to ATMega through a digital bus (SPI) that is used both when using the built in MAC layer of the radio (*burst mode*) or when transmitting bits directly (*direct mode*).

In either mode the configuration details of the nRF2401 is setup by uploading a *configuration* (or control word) to the device (see Figure 4.4). The configuration sets parameters such as receive/transmit mode, data-rate, etc. The control word is split in two parts one common part for direct-mode and ShockBurst and one only required for ShockBurst.

In the following we describe how to control the bus and how to setup the various modes of operation.

4.3.1 Mastering SPI

Communicating with the nRF2401 takes place over a three-wire serial interface not unlike SPI. The ATMega128l does not have such a peripheral, but the SPI unit is close enough to be useful.

In order to use the SPI peripheral of the ATMega128l it has to be misused slightly. The SPI interface is a four wire interface consisting of: chip select (CS), clock (CLK), master send (MOSI[1]) and slave send (MISO[2]). At each clock tick on the CLK line 2 bits are exchanged: one on the MOSI line and one on the MISO line. The ATMega128l has one register for each of these

[1]Master Output Slave Input
[2]Master Input Slave Output

lines — one outbound register and one inbound register. A transmission is initiated by putting a byte in the outbound register, starting the clock generator. Once the transmission is over the received data will reside in the inbound register.

In our case, the MOSI and MISO lines are combined since we never receive data from and transmit data to the nRF2401 the same time. However it is still possible to utilize the SPI interface of the ATMega128l. Sending works as described above, but in order to receive data from the nRF2401 we need the ATMega128l to start generating a clock without interfering with the signal from nRF2401. To do this we start the transmission of the byte “0x00”. Once the transmission of the “0x00” byte is completed, the value from the nRF2401 has been shifted into the inbound register.

The SPI peripheral can be operated either in a polling mode or using interrupts. Using the method above complicates the interrupt handling. Usually an interrupt is generated when a transmission is complete, however when we are trying to clock data out of the nRF2401 this must be handled properly.

The `nRFSPI` abstracts the bit level operations of communicating with the nRF2401, but allows full flexibility. ShockBurst and direct-mode operations communication is possible using this component.

4.3.2 SPI Master Mode and PB0

By mistake, PB0 ($\hat{\text{S}}$S not slave select) is connected to DR1 in the final platform, PB0 will be lifted off the board and, thus, be completely irrelevant, however in the following we describe the consequences of this mistake.

The DR1 signal is driven by the nRF2401 and the SPI peripheral of the ATMega is influenced by the state of this signal - particularly in SPI master mode. When communicating over the SPI bus the master shall generate a clock signal. The nRF2401 does not have a clock generator for this signal, thus the ATMega will always be designated as the master.

For the short burst mode, DR1 will toggle when data is ready. In direct mode, DR1 will be pulled low. For master mode of SPI peripheral, both of these are wrong. In the master mode, $\hat{\text{S}}$S is disregarded **only** if this is set to output (see [2, p.166]) - in this case both the ATMega and nRF2401 will attempt to drive the line. If PB0 is set as *input* (still, while the SPI is in the master mode) PB0 **must** be held high by an external source for the SPI bus to operate, which we cannot guarantee is performed by the nRF2401.

It is, thus, important that we are able to ignore the $\hat{\text{S}}$S value.

4.3.3 ShockBurst Mode

The Shock Burst mode provides a built in CRC check, MAC protocol, etc.The radio control data and data packet are sent via SPI.

Shock burst is enabled by uploading a special configuration word to the device (see Figure 4.4). From here on the radio handles the address match and CRC check and only returns valid packets. A shock burst TX could look like this:

1. AVR enbles the SPI bus as a master
2. AVR powers up the radio and waits for it to be ready
3. AVR sends a shock burst mode configuration to the nRF2401
4. AVR sends a packet with packet header to nRF2401

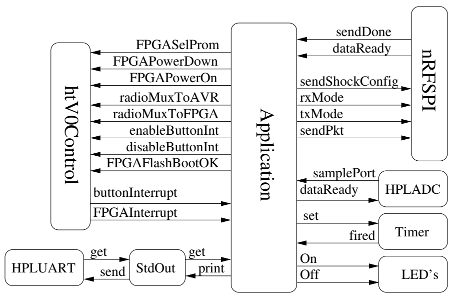


Figure 4.5 *TinyOS components*

4.3.4 Direct Mode

It should be possible to use the SPI unit to clock data in or out of the nRF2401. In order to do this the bit stream must be constant and without jitter. Any holes in the stream would be directly translated to the air. Eg. if the SPI unit is waiting for more bit the radio would stall. A direct mode session would look something like the following:

1. AVR enbles the SPI bus as a master
2. AVR sends a direct mode configuration to the nRF2401
3. AVR flips CE, CS appropriately and the nRF2401 enters the active mode

   The PB0 line will now be driven low constantly by the nRF2401 and has to be ignored.

   For both Tx and RX
4. It should be possible to clock data in and out of the nRF.

4.4 TinyOS

To span across multiple platforms easily TinyOS introduces the concept of platforms. For each platform implementations are provided for certain low-level interfaces used by higher level applications (one can think of these as drivers in a traditional operating system). By sharing interfaces across platforms an application can easily be compiled for multiple platforms. In our case we have implemented the HogthrobV0 platform, but for testing we are also using the BTnode2 platform.

4.4.1 Porting TinyOS

The core of TinyOS is very slim — it contains the simple thread model of TinyOS and little more. Subsequently, porting TinyOS is trivial, but in order to access the peripherals of the HogthrobV0 platform we must implement corresponding software components.

We implement `nRFSPI` and `htV0Control` to access the radio transceiver, the FPGA, the bus-exchange switches, and the push-buttons. For compatibility with existing TinyOS components (such as `IntOutput`, `IntDebug`, etc.) the LEDs are controlled through the `Leds` component. The component `htV0Control` provides control of the remaining components of the HogthrobV0 platform: FPGA, buttons.

In addition to this the ATMega128l based Mica variants and BTNode2 share the common meta platform *avrmote* — this platform provides only functionality related to the ATMega128l. The HogthrobV0 shares the same processor and we reuse as many components as possible.

4.4.2 FPGA, ATMega Interconnect

The FPGA is connected to the ATMega128l through the external memory interface — from the ATMega the FPGA is merely memory mapped to a special portion of memory.

The `htV0Control` components contains abstractions to enable and disable the external memory interface. In addition to the memory interface the two are connected with an interrupt line from the FPGA to the ATMega128l, `htV0Control` provides a TinyOS event for this interrupt.

This component alone does not provide the means for communicating between a processor core in the FPGA and the ATMega128l. This will have to be constructed as an additional component.

4.4.3 nRFSPI

The `nRFSPI` component work as a wrapper for the functionality of the nRF2401 and provides a *packet level* interface to the *byte level* SPI peripheral of the ATMega128l. The component is shared among the platforms that we are working with (BTNode2 and HogthrobV0). The nRF-SPI component is not a MAC, it assists in communicating with the nRF2401, but does not handle collisions and retransmissions or any other facilities that one would expect from a MAC. Furthermore, it does not handle the timing required when switching operation mode of the nRF2401, this will have to be implemented in an additional component.

The interface of `nRFSPI` abstracts the access of the nRF2401 by providing events and commands for common operations and providing data structures with human readable field names. The interface is shown in Figure 4.6 and the two data structures used by the interface is shown in Figure 4.7.

TinyOS does not provide any form of memory management. This means that the programs will have to keep track of the used and available space. The two common approaches to this in TinyOS are *transfer of ownership* and *buffer trading*[3]. Transfer of ownership implicitly transfers the ownership of a buffer when it is passed from component to component. It is up to the components to ensure that only the right one modifies it at the right time. Buffer trading denotes the process of giving a buffer to a component and getting one back. Using the data structures enables the buffer trading type of memory management — trading chunks of equal size. The two types of memory management are not mutually exclusive and can be used to the convenience of the programmer.

In the nRFSPI interface the event `dataReady` is an example of buffer trading while `rxMode` is an example of transfer of ownership.

```
interface nRFSPI {
 command void enableSPIMaster();

 /* Set up the nRF2401 in rx mode and provide a buffer for reception
  * The buffer must be atleast as big as ADDR_LEN and PAYLOAD_LEN
  *
  * The buffer is given back when the radio is set to txMode
  */
 command void rxMode(nRF_pkt_t* pkt);

 /* Set the nRF2401 in tx mode and give back a buffer given in rxMode
  * of NULL if no buffer was given.
  */
 command nRF_pkt_t* txMode();

 /* Send a payload of "pkt" to the recipent in "pkt".
  *
  * @return SUCCESS if no byte were in transit or a buffer
  *         was available.
  */
 command result_t sendPkt(nRF_pkt_t *pkt);

 command result_t sendShockConf(shock_conf_t *conf);
 command result_t sendBytesRev(uint8_t *first, uint8_t *last);
 async event result_t sendDone();

 /* Propagates data from the air to an application.
  * channel dennotes the transmission channel (1 or 2)
  * last signals the end of the current packet (DR1/DR2 low)
  */
 async event nRF_pkt_t* dataReady(uint8_t channel, nRF_pkt_t* pkt);

 /* Non-interrupt controlled interface */
 command void send_sync(uint8_t data);
```

Figure 4.6 *The TinyOS interface of nRFSPI*

```
typedef struct {
  uint8_t payload[PAYLOAD_LEN];
  uint8_t addr[ADDR_LEN];
}__attribute__((packed))nRF_pkt_t;
```

(a) Data packet structure

```
typedef struct {
   unsigned int      rx_en   : 1; // RX or TX operation
   unsigned int      rf_ch   : 7; // Channel frequency
   unsigned int      rf_pwr  : 2; // RF output power
   unsigned int      xo_f    : 3; // Crystal frequency
   unsigned int      rfdr_sb : 1; // RF data rate
   unsigned int      cm      : 1; // Direct/ShockBurst
   unsigned int      rx2_en  : 1; // Two channel receive
   //high order bits
 } __attribute__ ((packed))      gen_config_t;
```

(b) Common configuration structure

Figure 4.7 *Two data structures for the nRFSPI interface.*

Chapter 5

Testing

The goal of the following tests is to ensure that all the external interfaces (pin headers) and the on-board connections to the LED's, Push Buttons, and all the chip-to-chip interfaces are working properly. The tests are to be performed one time only for each board ensuring a uniform assurance for each board. We will focus on the chip-to-chip interfaces and assume that unless we detect errors with the following tests the components are working.

We will be assuming that no X-ray of the board will be performed meaning that there might be short circuits on the boards that we need to detect. However, such a short could be on the pins not connected to the pin headers or other chips. We will not be able to detect such errors and they will not influence the functionality of the platform.

Furthermore the tests are going to ensure that component mounting was accurate, and that no mistakes were made during post production modifications.

In the following we will describe the ATMega tests in detail and briefly cover the FPGA testing (FPGA testing is covered in detail in [4]).

5.1 AVR Testing

The test programs for the AVR will be written in the TinyOS and shared via the subversion repository at: https://svn.hogthrob.dk. If possible, the tests will be carried out using the on-board LED's and Push-Buttons. Before the test is started, the fuses MUST be programmed.

The upload port of the AVR will be tested first by uploading and downloading test patterns to the flash. Each of the following tests will be carried out by a single program uploaded to the AVR. The tests are carried out by connecting the board to a terminal emulator on a PC via the serial interface and an RS232 level-converter.

The tests are:

Echo Echo the typed character back to the user.

LED Turn the LEDs on motherboard and radio-board.

Button Notify a Push-Button press to the UART

ADC Print the value of the ADC to the UART

FPGA↔AVR Write patterns to the entire address space of the interface and report status via the UART

nRF2400 Set one node as RX and one as TX and try to make them communicate

The tests should be performed in the following order:

The test sets the Bus-Switches (MUX) to point towards the AVR at boot-up. If this fails, the UART1 LED's will not function.

5.1.1 Fuse programming

The fuse setup particular features of the ATMega (see section 4.1.1).

1. Use UISP to program fuses

5.1.2 Program upload

Test the program upload port (AVR-UART0, AVR→PEN).

1. Test that the PEN (program enable) and the UART0 (RXD0/RXD1) pins are connected and working.
2. Connect the HoghthrobV0 comm_port to the STK500 ISP.
3. Upload anything to the Flash and read it back using USIP (on Linux) The test patterns are: 0's, 1's and alternating 1's and 0's.
4. The uploaded and the downloaded programs should be the same.

A simple way to test this is simply to test this is to supply uisp with the "–verify" option:

```
    uisp --verify -dprog=stk500 -dserial=/dev/ttyUSB0 -dpart=ATmega128 -v=2 --erase
--upload if=build/hogthrobV0/main.srecb
```

5.1.3 ATMega UART1

Test the connection to the secondary UART (AVR-UART1)

1. Test that the Rx and the Tx lines are connected to the Pin Header (Radio_IO). Test the buffer (NC7SX125) and the control line (Radio_IO11 → PC11)
2. Connect RXD1/TXD1 on the Radio Connector to the PC via RS232 level- converter (on STK500). Upload the test program to the AVR. Start MiniCom terminal emulator on PC (with Linux) or a similar program with Windows (Hyperterminal). Local echo should be disabled in the terminal emulator. The control line for NC7SX125 is set 'High' and the resistor R37 is unmounted (if mounted).
3. Each typed character should be echoed back immediately.

5.1.4 ATMega LED

Test connection to LEDs (D0 on motherboard and D0/D1 on radio board)

1. Test that the LED's on the MB (LED → PD7) and on the Radio Board (RADIO_IO7/RADIO_IO8 → PC3/PG4) are working
2. Start the blink test in the test program
3. See the LED's blinking

5.1.5 ATMega push-buttons

Test the ATMega push button (AVR - Push-Button).

1. Test that the on-board Push-Button is connected to the AVR (PE7)
2. Upload the test program (the Push-Button test is enabled by default)
3. For each Push-Button press, the UART should report this.

5.1.6 ATMega radio connection and bus switches

Test connection to the radio, this connection goes through the bus switches (AVR-nRF). Test that the Radio Transceiver is able to communicate through the connector to the Radio Board. Test the MUX and the MUX-control lines (RADIO_MUX0/1/2 → PC4/PC5/PC6).

1. Use 2 nodes. Connect the Radio Boards to the MB. Upload the test program.
2. The test program has a one-way test (the receiver tests the CRC). It sets the the MUX to connect the radio and the AVR. Start by enabling one as an RX, and then enable the other as a TX
3. As soon as the TX-node is enabled, the RX-node should start blinking the LED's and print status on the terminal.

5.1.7 ATMega sensor connector

Test the external sensor connections. These connection can function either as analog or digital, we only test their analog mode.

1. Test that the ADC0-ADC7 connections to J3 are working properly.
2. Connect all the sensors to ground (pin 1-8 to pin 15) and to AREF reference voltage (pin 1-8 to pin 9), respectively.
3. Start the ADC printer on the node
4. With ADC input grounded, all should be 0 and with all set to AREV they should be 0xFFFF.

5.2 FPGA Testing

The FPGA will be tested using a simple VHDL state machine. This state machine will input and output some registers - continuously stimulating the inputs and the outputs. The tests will be carried out using the on-board LED's and connecting a logic-analyzer and capturing the waveforms. The Logic Analyzer can be replaced with the Xilinx ChipScope. FPGA testing is further documented in [4].

Required Programs:

AVR FPGA-bootup-and-MUX-selector, memory write-read-back-test

FPGA Xilinx ChipScope core, AVR-SRAM interface, FPGA→FLASH interface

5.2.1 PROM Programming (Upload)

Test JTAG and PROM-lines.

1. Connect JTAG and upload a configuration.
2. Read it back and see that the two are the same.

5.2.2 FPGA Boot (FPGA control-lines)

Test that the FPGA↔AVR control-lines are correct and that the FPGA boots.

1. Upload the AVR program.
2. AVR sets the correct PROM () and powers up the FPGA (power_on) - FPGA loads the program from the PROM. Set PROM_SEL (It will be pulled-up now) AVR wait for the DONE Signal. Turn on the LED's.
3. After applying power, the AVR-connected LED (on the MB) should turn on after "a while".

5.2.3 FPGA→LED, Push-Button

1. Test the LED and the Push-Button connections using the Xilinx ChipScope
2. Upload FPGA configuration and set the LED values in the ChipScope. Toggle the Push Buttons and see the changes.

5.2.4 FPGA→Sensor Board (Digital Connectors), FPGA→nRF, FPGA→UART (Serial Interface)

All of the following tests can be performed either manually using the Xilinx ChipScope or by writing a simple FPGA configuration that automates the procedure.

1. Test that the connections to the pin headers are working correctly using the Xilinx ChipScope.

   Connect both the Digital Connectors with each other (J4 to J5).

   If the Radio Transceiver is working with the AVR, we just need to test that the FPGA connections to the Radio Connector are working. Connect the pins on J1: 1,3,5,7,9,11,13,15→17,19,21,23,25,27,2
   After booting the FPGA, the AVR sets the MUX and the Tri-State Buffer.

   Short circuit Rx and Tx. Use the Xilinx ChipScope such that whatever you send, you get back.

2. Upload FPGA configuration with the Xilinx ChipScope. Generate test patterns on each of the I/O pin groups and check that they show up at the inputs. The patterns include 0's, 1's and alternating 1 and 0.

   See that the pattern from J4 shows up at J4, Rx→Tx and the J1 pin groups.

   To test the LED's, toggle the values in the Xilinx ChipScope

5.2.5 AVR→FPGA

Test that all the data and the control lines are working and that the FPGA can interrupt the AVR.

The FPGA is connected to the AVR using the external SRAM interface (p. 26 in the AVR data sheet). Reading and writing to a register in the FPGA will test all the pins (AD, ALE, RD_N, WE_N). The latch and the register are implemented in the FPGA and the AVR writes and reads this register.

Since only the lower 8 bits of the address are connected, the high bits of the address space must be disabled by setting XMM0=XMM1=XMM2=1. The XMEM interface will be configured to no wait-states, XMCRA=SRW10=0. The bus-keeper is enabled by setting XMBK=1.The external memory interface is enabled by setting SRE in MCUCR.

The FPGA is memory-mapped to the addresses 0x1100-0x90FF (both inclusive). The test program will write and read back a series of patterns (0's, 1's and alternating '1' and '0').

FPGA_CS is untested!!

1. Upload the test program.
2. Boot the FPGA and wait for the DONE signal.
3. Start the test program and wait for it to report success or failure.

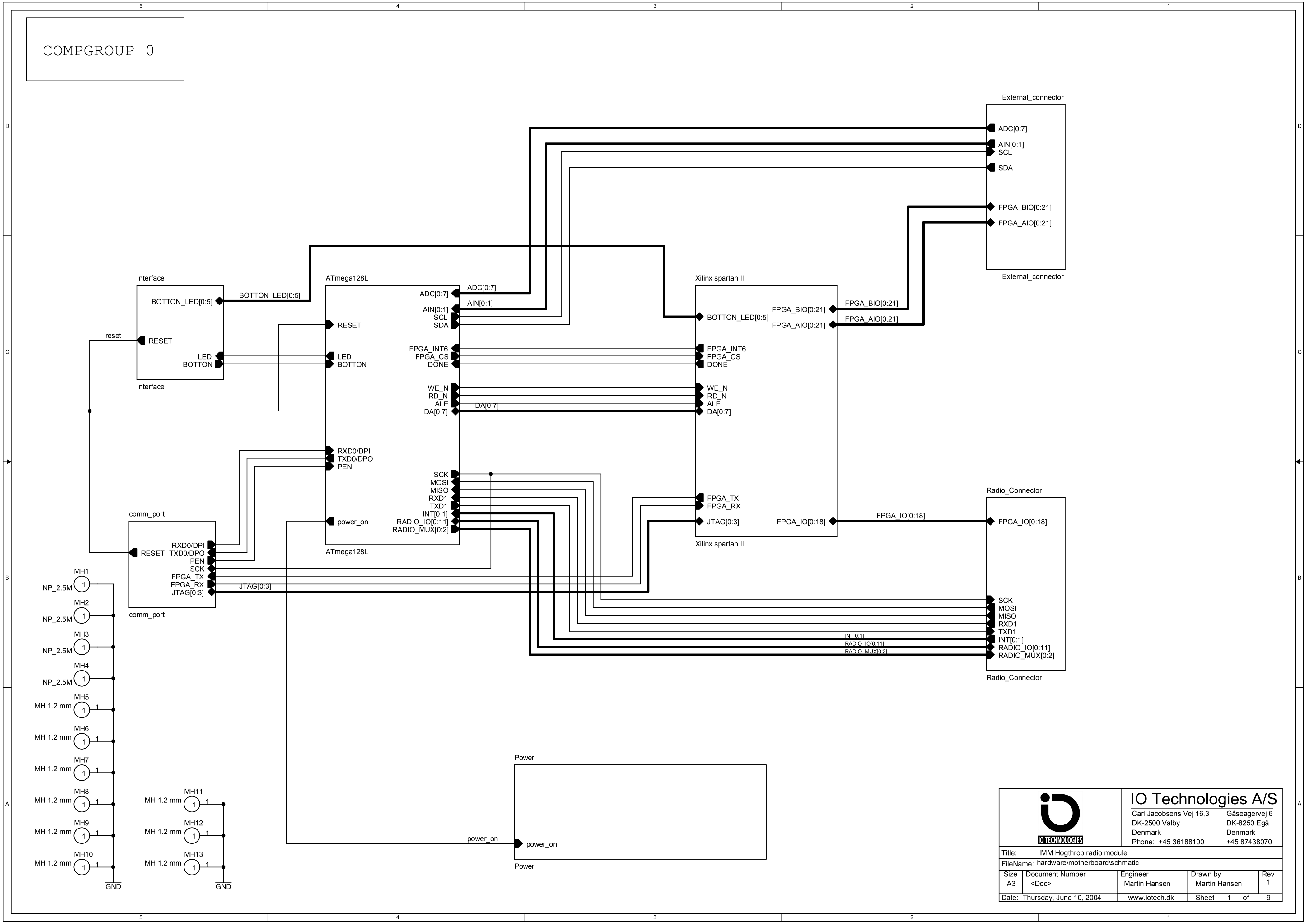
5.2.6 FPGA→FLASH

Make sure that the connection to the on-board flash is working. We don't need to test that the FLASH is working. The FPGA implements a simple serial interface to the flash and uploads some data and tries to read it back.

1. Try all 1's and all 0's, and alternating 1's and 0's.

Appendix A

Schematics



IO TECHNOLOGIES

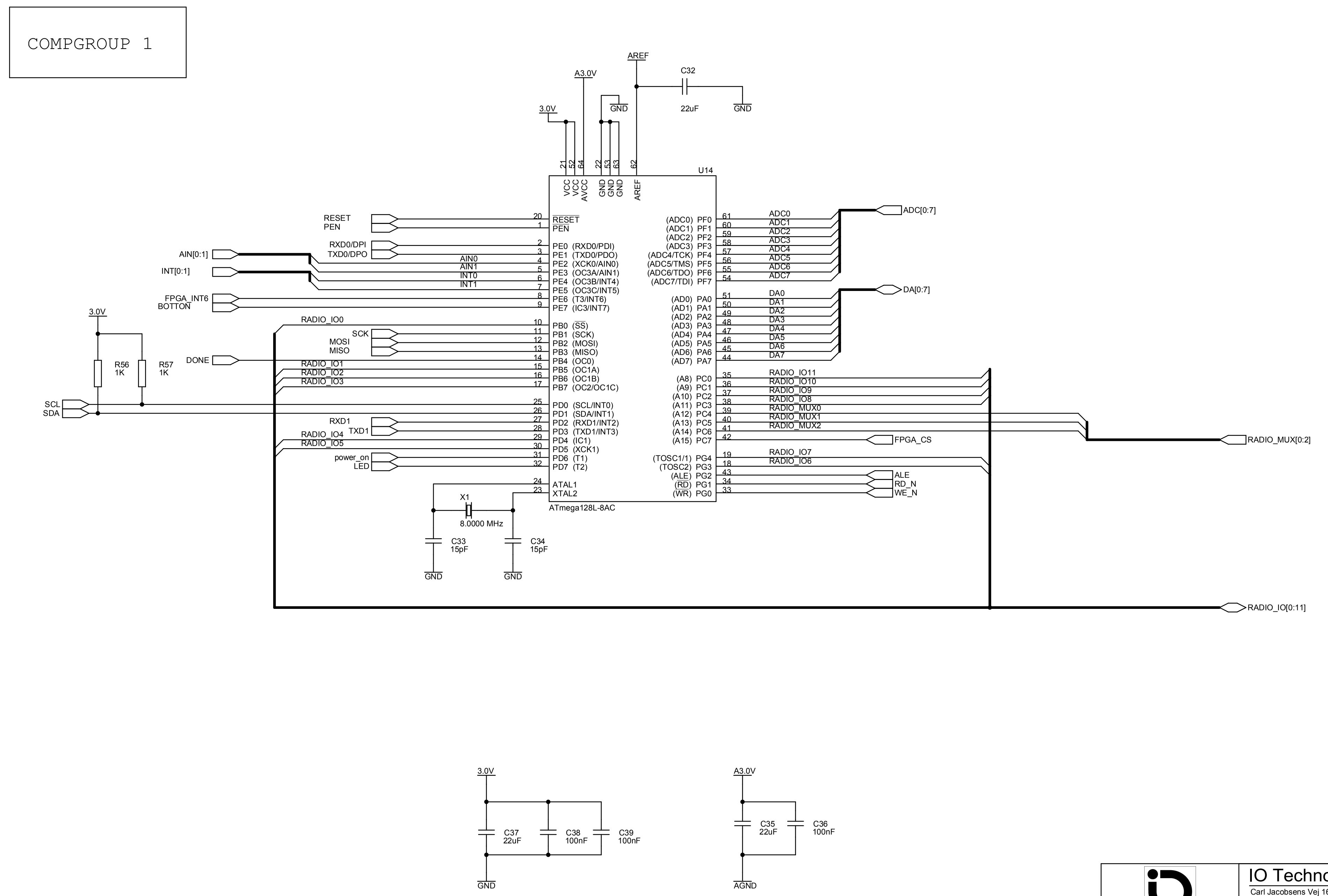
IO Technologies A/S

Carl Jacobsens Vej 16,3	Gåseagervej 6
DK-2500 Valby	DK-8250 Egå
Denmark	Denmark
Phone: +45 36188100	+45 87438070

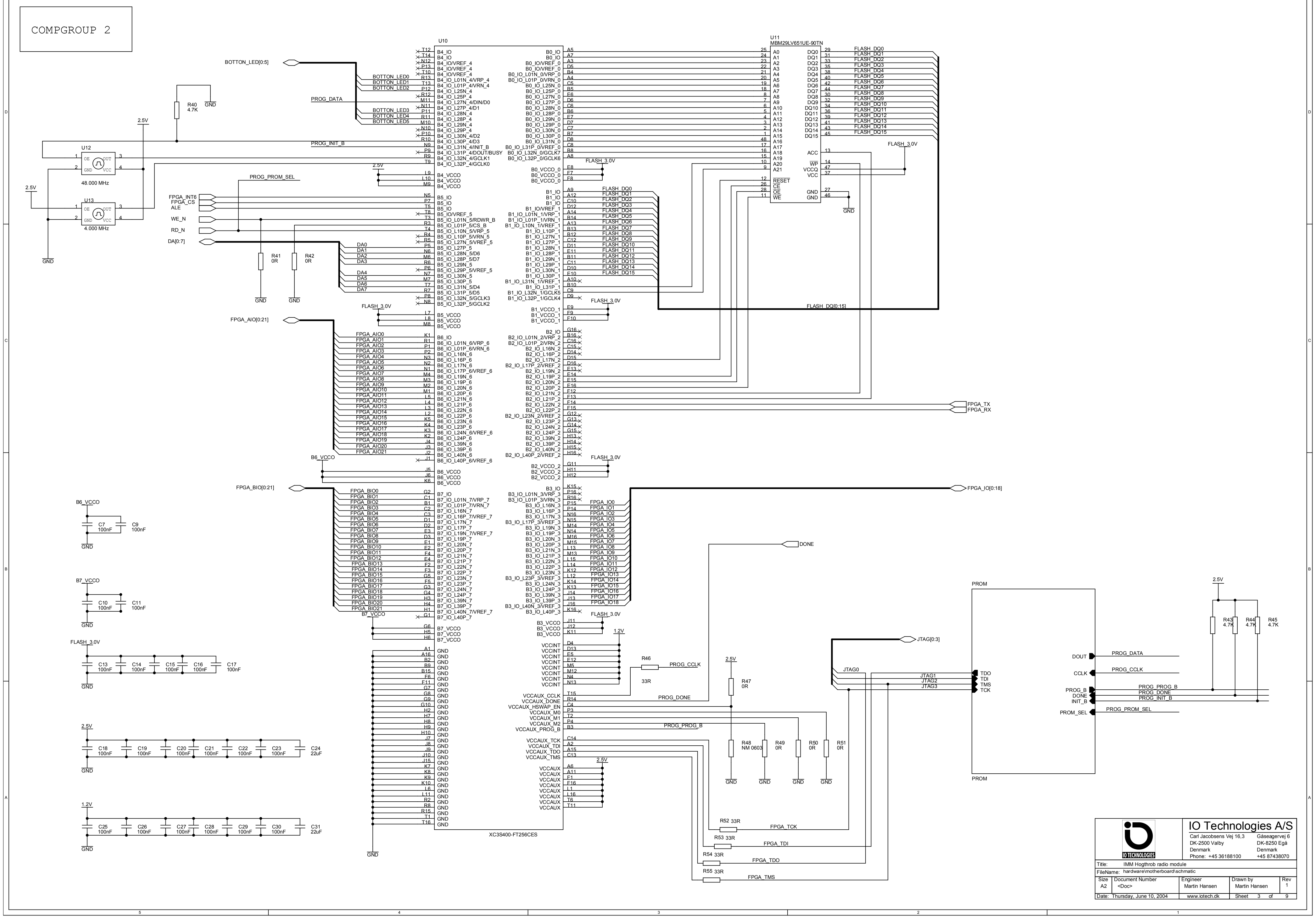
Title: IMM Hogthrob radio module

FileName: hardware\motherboard\schmatic

Size	Document Number	Engineer	Drawn by	Rev
A3	<Doc>	Martin Hansen	Martin Hansen	1
Date: Thursday, June 10, 2004		www.iotech.dk	Sheet 1 of 9	



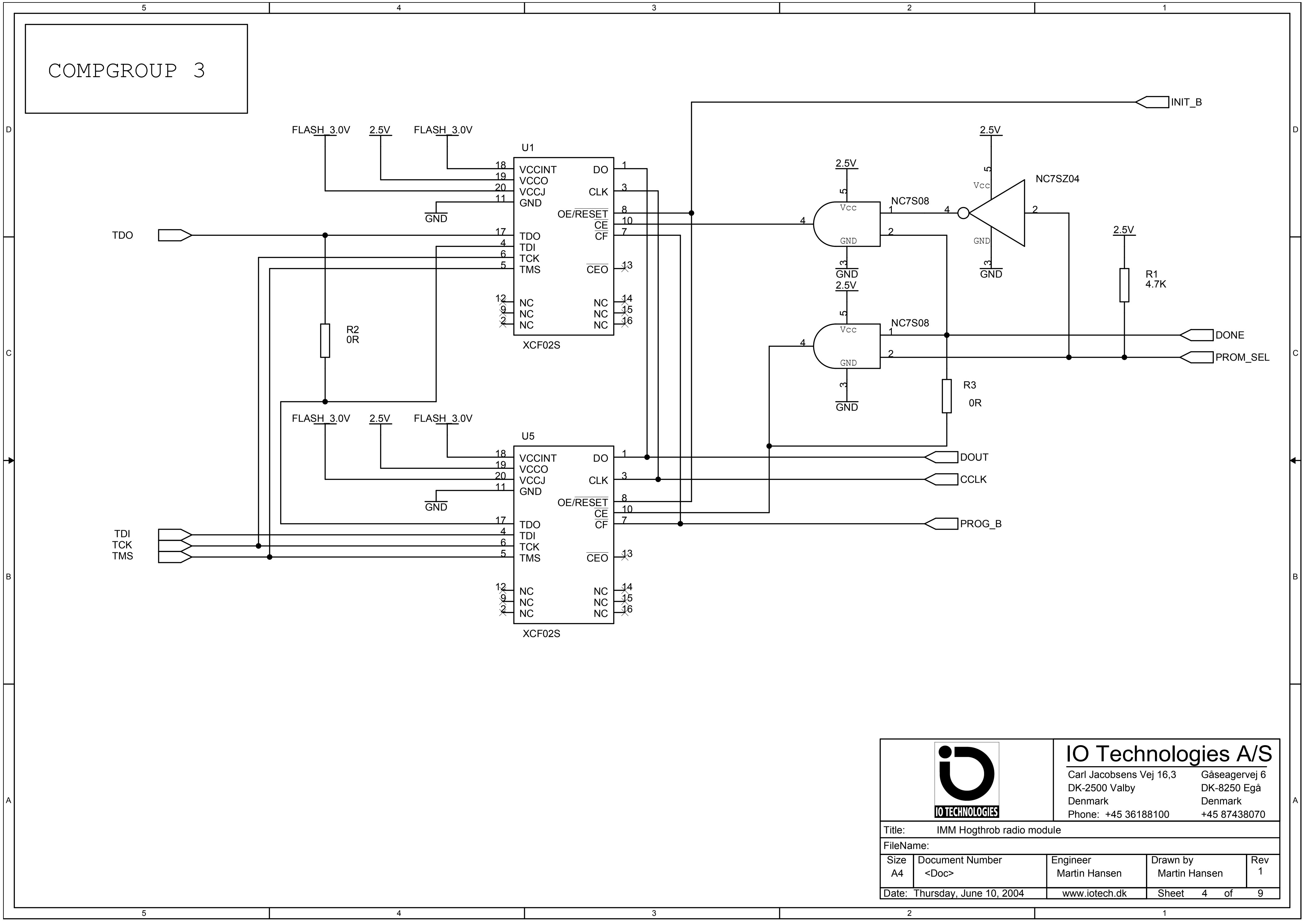
IO Technologies A/S				
Carl Jacobsens Vej 16,3 DK-2500 Valby Denmark Phone: +45 36188100	Gåseagervej 6 DK-8250 Egå Denmark +45 87438070			
Title: IMM Hogthrob radio module				
FileName: hardware\motherboard\schmatic				
Size A3	Document Number <Doc>	Engineer Martin Hansen	Drawn by Martin Hansen	Rev 1
Date: Thursday, June 10, 2004		www.iotech.dk	Sheet 2 of 9	



IO Technologies A/S

Carl Jacobsens Vej 16,3 — Gåsagervej 6
DK-2500 Valby — DK-8250 Egå
Denmark — Denmark
Phone: +45 36188100 — +45 87438070

Title: IMM Hogthrob radio module			
FileName: hardware\motherboard\schmatic			
Size A2	Document Number <Doc>	Engineer Martin Hansen	Drawn by Martin Hansen	Rev 1
Date: Thursday, June 10, 2004		www.iotech.dk	Sheet 3 of 9



IO Technologies A/S

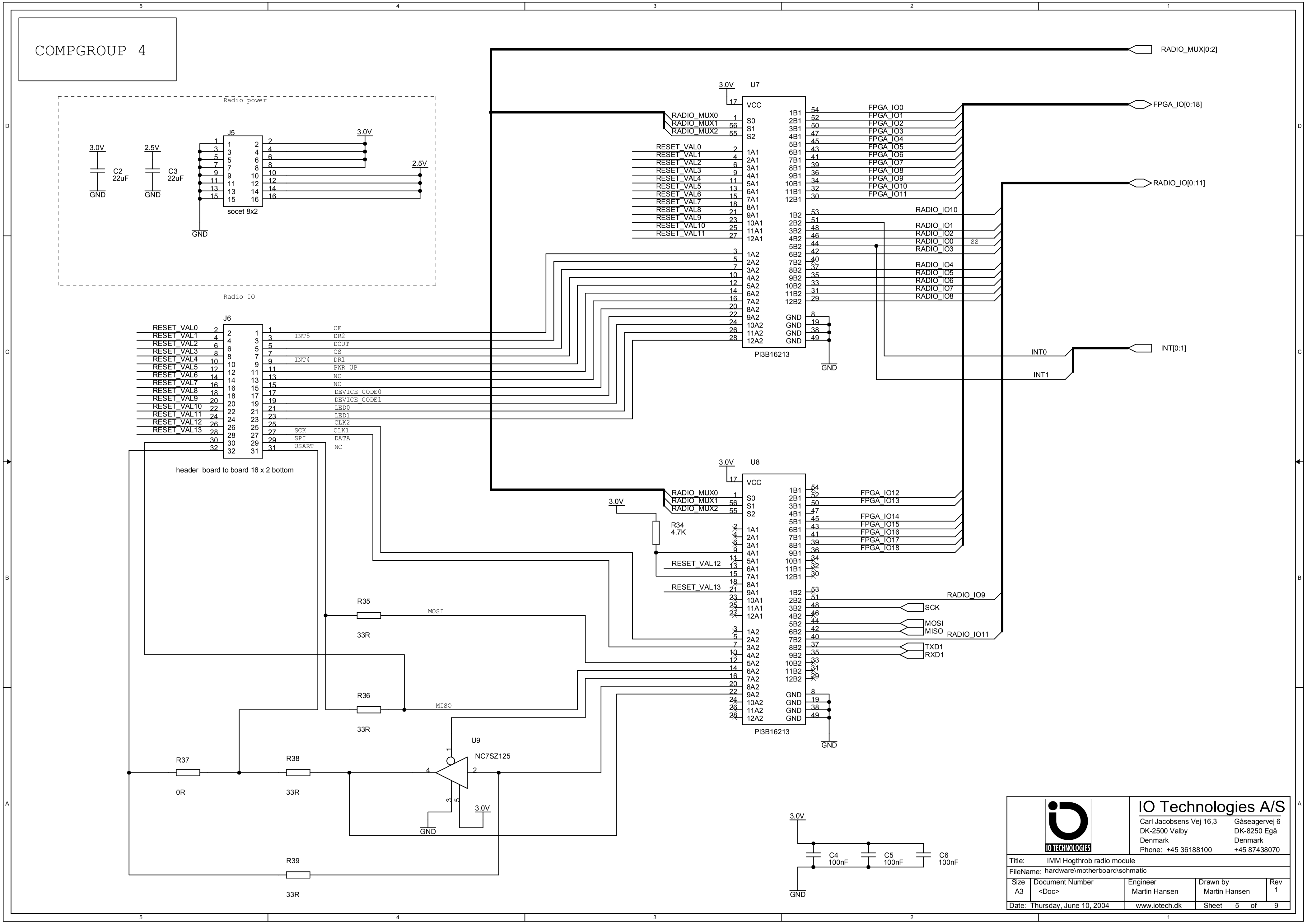
Carl Jacobsens Vej 16,3	Gåseagervej 6
DK-2500 Valby	DK-8250 Egå
Denmark	Denmark
Phone: +45 36188100	+45 87438070

Title: IMM Hogthrob radio module

FileName:

Size	Document Number	Engineer	Drawn by	Rev
A4	<Doc>	Martin Hansen	Martin Hansen	1

Date: Thursday, June 10, 2004 | www.iotech.dk | Sheet 4 of 9

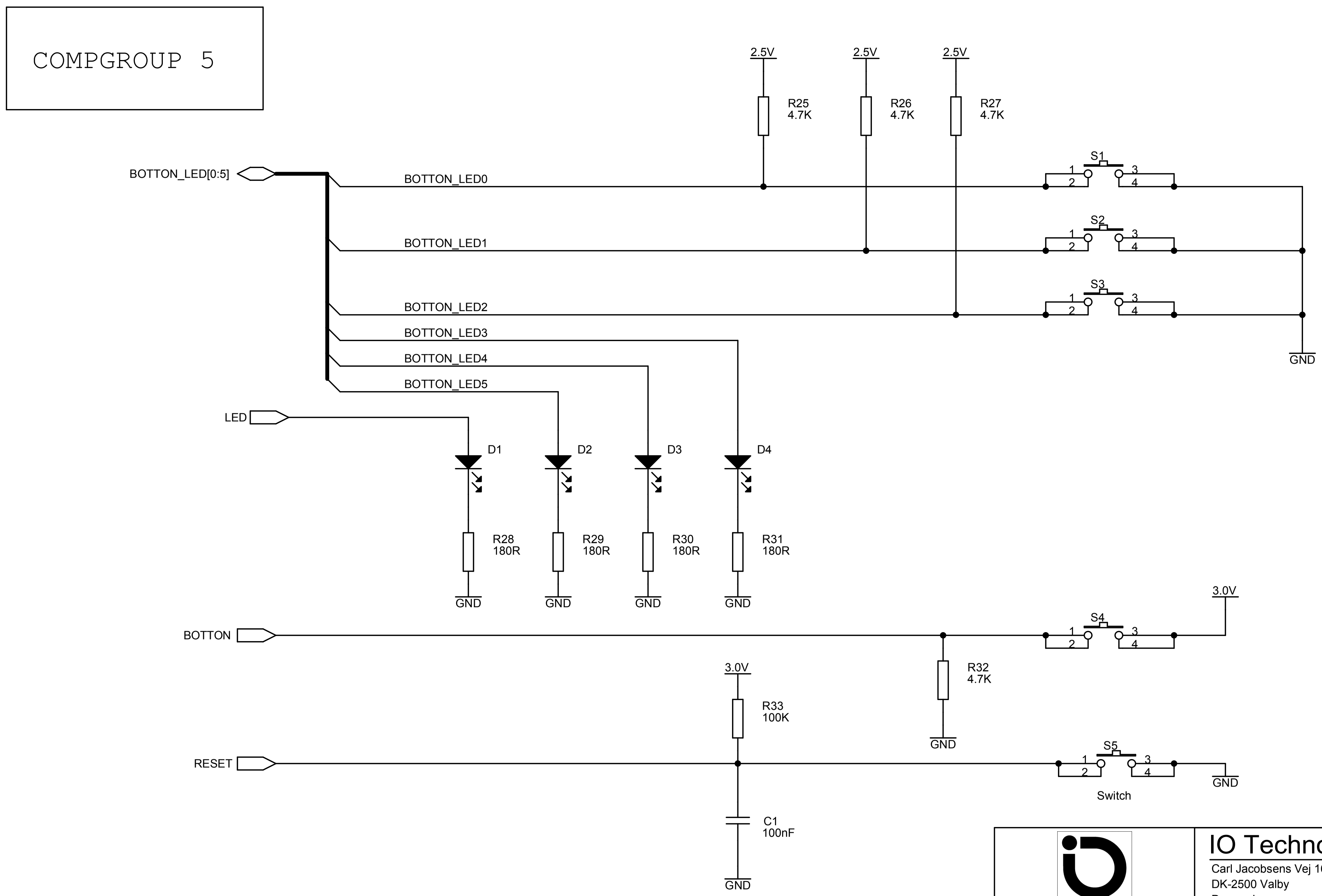


IO Technologies A/S
Carl Jacobsens Vej 16,3
DK-2500 Valby
Denmark
Phone: +45 36188100

Gåseagervej 6
DK-8250 Egå
Denmark
+45 87438070

IO TECHNOLOGIES

Title: IMM Hogthrob radio module				
FileName: hardware\motherboard\schmatic				
Size A3	Document Number <Doc>	Engineer Martin Hansen	Drawn by Martin Hansen	Rev 1
Date: Thursday, June 10, 2004		www.iotech.dk	Sheet 5 of 9



IO TECHNOLOGIES

IO Technologies A/S

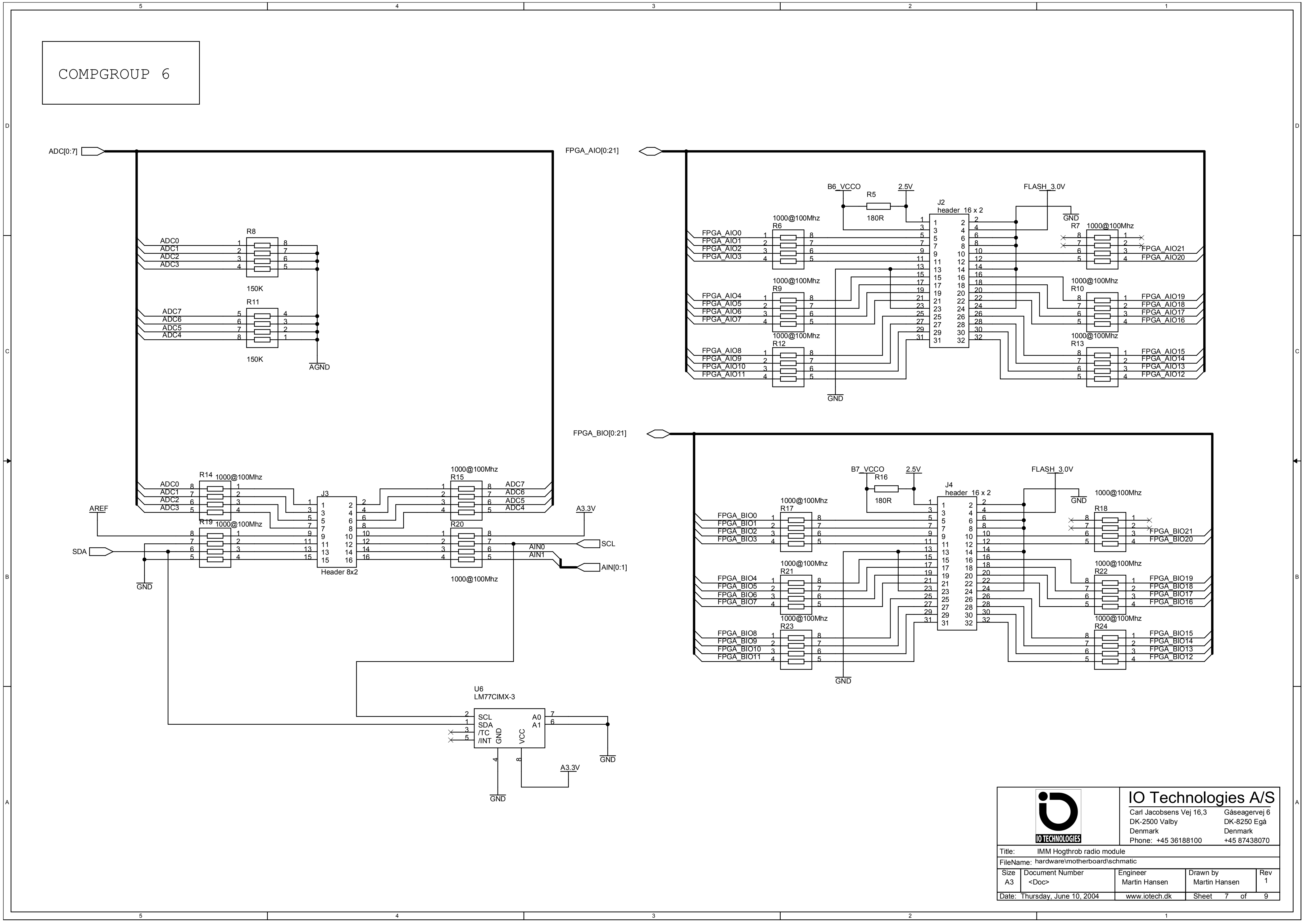
Carl Jacobsens Vej 16,3	Gåseagervej 6
DK-2500 Valby	DK-8250 Egå
Denmark	Denmark
Phone: +45 36188100	+45 87438070

Title: IMM Hogthrob radio module

FileName:

Size	Document Number	Engineer	Drawn by	Rev
A4	<Doc>	Martin Hansen	Martin Hansen	1

Date: Thursday, June 10, 2004 | www.iotech.dk | Sheet 6 of 9



IO Technologies A/S

Carl Jacobsens Vej 16,3	Gåseagervej 6
DK-2500 Valby	DK-8250 Egå
Denmark	Denmark
Phone: +45 36188100	+45 87438070

IO TECHNOLOGIES

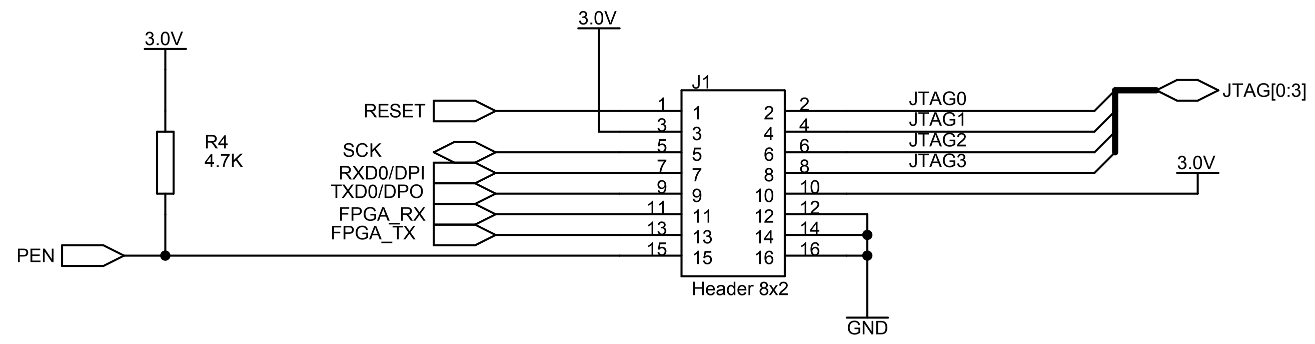
Title: IMM Hogthrob radio module

FileName: hardware\motherboard\schmatic

Size	Document Number	Engineer	Drawn by	Rev
A3	<Doc>	Martin Hansen	Martin Hansen	1

Date: Thursday, June 10, 2004 | www.iotech.dk | Sheet 7 of 9

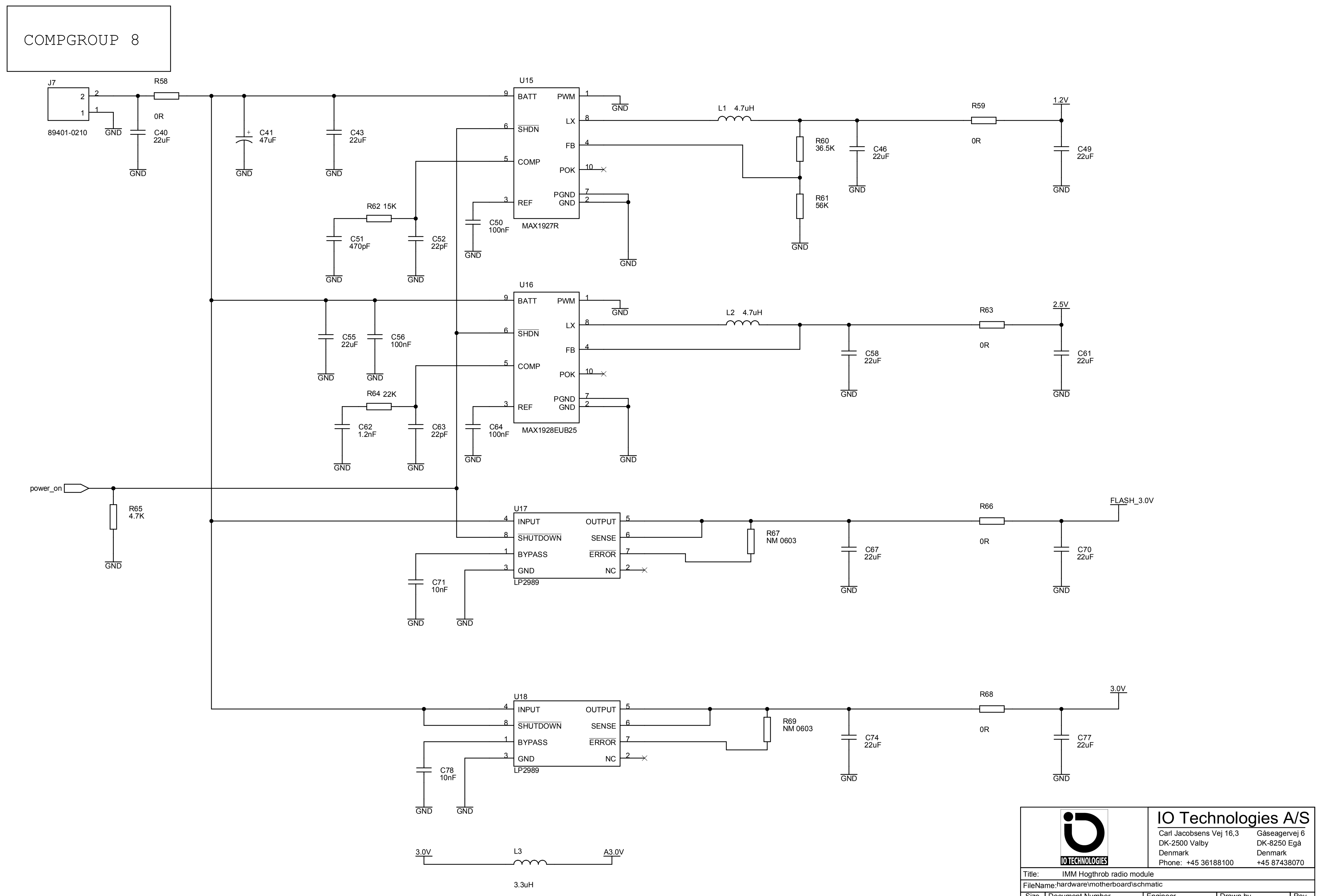
COMPGROUP 7



IO Technologies A/S

Carl Jacobsens Vej 16,3	Gåseagervej 6
DK-2500 Valby	DK-8250 Egå
Denmark	Denmark
Phone: +45 36188100	+45 87438070

Title: IMM Hogthrob radio module				
FileName:				
Size A	Document Number <Doc>	Engineer Martin Hansen	Drawn by Martin Hansen	Rev 1
Date: Thursday, June 10, 2004		www.iotech.dk	Sheet 8 of 9	



IO TECHNOLOGIES

IO Technologies A/S

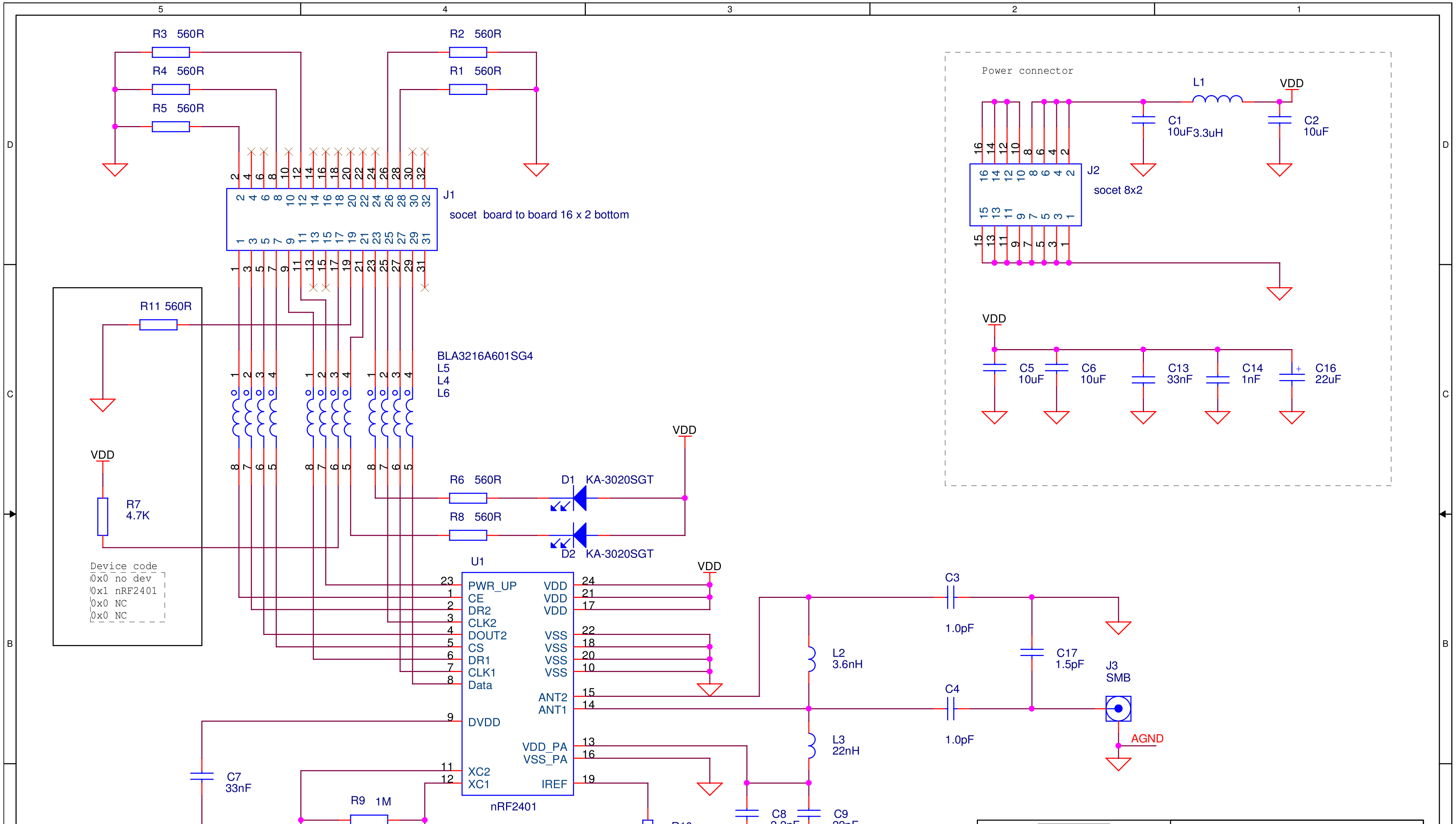
Carl Jacobsens Vej 16,3	Gåseagervej 6
DK-2500 Valby	DK-8250 Egå
Denmark	Denmark
Phone: +45 36188100	+45 87438070

Title: IMM Hogthrob radio module

FileName: hardware\motherboard\schmatic

Size	Document Number	Engineer	Drawn by	Rev
A3	<Doc>	Martin Hansen	Martin Hansen	1

Date: Thursday, June 10, 2004 www.iotech.dk Sheet 9 of 9



IO Technologies A/S

Carl Jacobsens Vej 16,3
DK-2500 Valby
Denmark
Phone: +45 36188100

Gåseagervej 6
DK-8250 Egå
Denmark
+45 87438070

Title: IMM Hogthrob radio module				
FileName: hardware\radio_module\schmatic				
Size A4	Document Number <Doc>	Engineer Martin Hansen	Drawn by Martin Hansen	Rev 1
Date: Wednesday, April 21, 2004		www.iotech.dk	Sheet 1 of 1	

Appendix B

Errata

This chapter contains an errata from the HogthrobV0 Platform and Associated Documentation.

- LED on MB and comm. board have different semantics
- $\hat{S}$S connected wrongly (should be unconnected)
- Sensing and radio boards are connected to different sides of the MB PCB
- J3 - 3.3 V measures as unconnected
- built in battery-voltage tester missing
- On-MB Temperature Sensor missing
- The signals PROM_SEL, FPGA_CS and PROG_PROM_SEL are confusing FPGA_CS is connected from AVR→FPGA (is this a general purpose pin??) PROM_SEL is depicted in comgroup 3, but not connected to AVR (could be multiplexed with RD_N - PG1) PROG_PROM_SEL is only shown in comgroup 2 nowhere else PROG_PROM_SEL is connected to RD_N (AVR)

  Possibly RD_N acts as PROM_SEL during FPGA boot and laters as ext. mem interface meaning that FPGA_CS is just an I/O pin between AVR/FPGA
- The 2nd UART has been routed to the radioboard with no purpose. Furthermore a criscrossing cuircuit has been placed in front of it making it useless as a debug port for the ATMega.

B.1 Post Delivery Modifications

In order to make the platform boot the following modifications have been made:

- Removed (Comgroup 2) R41, R47 (on some boards), R48
- Removed (Comgroup 3) R3
- Removed (Comgroup 4) R34, R37
- Removed (Comgroup 8): R67, R69
- On some boards R36 (comgruop 4) has been removed. This should be replaced.

B.2 Bill of Components

- L3 might be changed to Murata LQH32CH15M11l (check index.htm)
- The actual FLASH memory for the FPGA is unclear (stykliste and index.html are inconsistent)

B.3 Schematics

The schematic is out of date with the actual layout in several places. These mistakes refer to the schematic dated June 10 2004.

- - C37, C77 missing
- Commgroup Group 1:
  - ATAL1 is named XTAL1
  - BUTTON misspelled BOTTON
- Commgroup Group 2:
  - "Done" is an output
  - FPGA_INT6 is output
  - U11 is depicted as a different chip than the one that is mounted,
  - VCCO_3-VCCO_7 are not named consistently with VCCO_0-VCCO_2
- Comgroup 3:
  - U3 and U4 are not designated on the schematic (U3 is the NC7S08 on the top and U4 is the one on the bottom).
- Comgroup 4:
  - INT4/INT5 on Datasheet wrong

Appendix C

FPGA_control.c

```
#define __AVR_ATmega128__ 1

#define FALSE 0
#define TRUE 1

#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>


void FPGA_on();
void toggleFPGA();
void setMuxToAVR();

int FPGA_running, muxToAVR, ledsOn;


#define TOSH_ASSIGN_PIN(name, port, bit) \
static inline void TOSH_SET_##name##_PIN()      {PORT##port |=  _BV(bit);}\
static inline void TOSH_CLR_##name##_PIN()      {PORT##port &= ~_BV(bit);}\
static inline void TOSH_MAKE_##name##_OUTPUT() {DDR##port  |=  _BV(bit);}\
static inline void TOSH_MAKE_##name##_INPUT()  {DDR##port  &= ~_BV(bit);}\
static inline char TOSH_READ_##name##_PIN()     {return 0x01 & (PIN##port >> bit);}


// Setup the nRF mux control lines
TOSH_ASSIGN_PIN(nRFMUX_s0, C, 4);
TOSH_ASSIGN_PIN(nRFMUX_s1, C, 5);
TOSH_ASSIGN_PIN(nRFMUX_s2, C, 6);

/* The uart. Uart0 is connected to the bluetooth module
   via pe0 and pe1. Uart1 is external, via pd2 and pd3. */
TOSH_ASSIGN_PIN(UART_RXD0, E, 0);
TOSH_ASSIGN_PIN(UART_TXD0, E, 1);

TOSH_ASSIGN_PIN(UART_RXD1, D, 2);
TOSH_ASSIGN_PIN(UART_TXD1, D, 3);

/* The FPGA is connected to the AVR via the external memory
   interface (only the lower address pins) and an interrupt
   pin */
TOSH_ASSIGN_PIN(WE_N, G, 0);
TOSH_ASSIGN_PIN(RD_N, G, 1);
TOSH_ASSIGN_PIN(ALE, G, 2);

TOSH_ASSIGN_PIN(FPGA_CS, C, 7);
TOSH_ASSIGN_PIN(FPGA_ON, D, 6);
TOSH_ASSIGN_PIN(FPGA_DONE, B, 4);

/* FIXME: REMEBER TO SET DDR AS INPUT!!! */
//TOSH_ALIAS_INT(BUTTON, INT7); // Dosn't compile!? INT7 is a macro!?
TOSH_ASSIGN_PIN(BUTTON_PIN, E, 7);
#define SIG_BUTTON SIG_INTERRUPT7
#define BUTTON_INT_ENABLE EIMSK |= _BV(INT7)
#define BUTTON_INT_DISABLE EIMSK &= ~_BV(INT7)
#define BUTTON_INT_CLR EIFR |= _BV(INT7) // Clear any leftover ints
#define BUTTON_INT_SETUP EICRB |= _BV(ISC71) | _BV(ISC70) // Int on rising edge

// Setup LED names, for compatibility we use the same names as Mica
// LedDebug (and IntOutput) uses the bitorder (MSB->LSB)
//  EXTRA_LED LED_YELLOW LED_GREEN LED_RED
//  0x8        0x4         0x2        0x1
//
// The LEDs on the radio board have reverse semeantics
// TOSH_CLR means on TOSH_SET means off (LedC handles this)
//
// Remember that LedDebug reverses the code!
TOSH_ASSIGN_PIN(MB_LED, D, 7);     // D1 on mother board
TOSH_ASSIGN_PIN(RadioD1_LED, C, 3);  // D1 on radio board
TOSH_ASSIGN_PIN(RadioD2_LED, G, 4); // D2 on radio board

  /*********************************************************
   *                                                       *
   *                    Boot FPGA                          *
   *                                                       *
   *                                                       *
   *********************************************************/

  void FPGA_on(){
    TOSH_MAKE_FPGA_ON_OUTPUT();
    TOSH_MAKE_FPGA_CS_OUTPUT();
    TOSH_SET_FPGA_ON_PIN(); // Actually boot
  TOSH_CLR_FPGA_CS_PIN(); // Doesn't matter
  while( ! TOSH_READ_FPGA_DONE_PIN()){};
}

void FPGA_off(){
  TOSH_MAKE_FPGA_ON_OUTPUT();
  TOSH_MAKE_FPGA_CS_OUTPUT();
  TOSH_CLR_FPGA_ON_PIN();
  TOSH_CLR_FPGA_CS_PIN();
}

void toggleFPGA(){
  if (FPGA_running) {
    FPGA_running = FALSE;
    FPGA_off();
  } else {
    FPGA_running = TRUE;
    FPGA_on();
  }
}


void setMuxToAVR() {
  DDRC |= _BV(2);
  DDRC |= _BV(3);
  DDRC |= _BV(4);
  DDRC |= _BV(5);
  DDRC |= _BV(6);
  DDRC |= _BV(7);

  PORTC &= ~_BV(4); //cbi(PORTC, 4);
  PORTC |=  _BV(5); //sbi(PORTC, 5);
  PORTC |=  _BV(6); //sbi(PORTC, 6);
  PORTC |=  _BV(7); //sbi(PORTC, 7);
  PORTC |=  _BV(8); //sbi(PORTC, 8);
}

void setMuxToFPGA() {
  //sbi(PORTC, 4);
  //sbi(PORTC, 5);
  //sbi(PORTC, 6);
  PORTC |=  _BV(4); //sbi(PORTC, 4);
}

void togglenRFMUX() {
  if (muxToAVR) {
    setMuxToFPGA();
    muxToAVR=FALSE;
  } else {
      setMuxToAVR();
      muxToAVR=TRUE;
    }
  }



  /*******************************************************
   *                                                     *
   *                     LED TEST                        *
   *                                                     *
   *                                                     *
   *******************************************************/

  // Remeber .debug reverses the code so 0 means all on!
  void toggleLeds(){
    if (ledsOn) {
      ledsOn = FALSE;
      TOSH_CLR_MB_LED_PIN();
      TOSH_SET_RadioD1_LED_PIN();
      TOSH_SET_RadioD2_LED_PIN();

    } else {
      ledsOn = TRUE;
      TOSH_SET_MB_LED_PIN();
      TOSH_CLR_RadioD1_LED_PIN();
      TOSH_CLR_RadioD2_LED_PIN();
    }
  }


  /*******************************************************
   *                                                     *
   *                Pin direction setup                  *
   *                                                     *
   *                                                     *
   *******************************************************/


void TOSH_SET_PIN_DIRECTIONS(void)
{
  DDRC=0x0; // PortC out
  PORTC & _BV(4); // Mux to AVR
  PORTC |= _BV(5);
  PORTC |= _BV(6);

  TOSH_MAKE_nRFMUX_s0_OUTPUT();
  TOSH_MAKE_nRFMUX_s1_OUTPUT();
  TOSH_MAKE_nRFMUX_s2_OUTPUT();
  TOSH_MAKE_MB_LED_OUTPUT();
  TOSH_MAKE_RadioD1_LED_OUTPUT();
  TOSH_MAKE_RadioD2_LED_OUTPUT();

  // Start with LEDs on
  ledsOn=TRUE;
  TOSH_SET_MB_LED_PIN();
  TOSH_CLR_RadioD1_LED_PIN();
  TOSH_CLR_RadioD2_LED_PIN();

  // It is extremely important that the pins going to the FPGA
  // are "tristated" while the FPGA is off or we risk connecting
  // an outputpin driving the line directly to ground
  //
  // All of the directions will be taken over by the memory
  // interface when enabled
  DDRA = 0xFF; // All in
  TOSH_MAKE_ALE_INPUT();
  TOSH_MAKE_WE_N_INPUT();
  TOSH_MAKE_RD_N_INPUT();

  // Turn FPGA off
  TOSH_MAKE_FPGA_ON_OUTPUT();
  TOSH_MAKE_FPGA_CS_OUTPUT();
  TOSH_MAKE_FPGA_DONE_INPUT();
  TOSH_CLR_FPGA_ON_PIN(); // Off

  TOSH_MAKE_BUTTON_PIN_INPUT();
}

int main(void) {
  TOSH_SET_PIN_DIRECTIONS();

  FPGA_running = TRUE;
  FPGA_on();

  muxToAVR = FALSE;
  setMuxToFPGA();

  return 0;
}
```

Appendix D

example.ucf

```
# Clock interfaces
NET "clk_40_en\"          LOC = "R10" ; # Clock enable for 40 MHz clock
NET "clk_40mhz"          LOC = "r9"   ; # 40 MHz clock
NET "clk"                LOC = "t9"   ; # Slow clock

# FPGA UART interface (J?)
NET "all_rxd_i<0>"       LOC = "f15"  ;
NET "all_txd_o<0>"       LOC = "f14"  ;

# Button interface
NET "reset"              LOC = "p12"  ;
NET "btn<0>"  LOC = "p12"  ;
NET "btn<1>"  LOC = "t13"  ;
NET "btn<2>"  LOC = "r13"  ;

# Led interface
NET "led<0>"  LOC = "m10"  ;
NET "led<1>"  LOC = "r11"  ;
NET "led<2>"  LOC = "p11"  ;


# ATMega <-> FPGA interface

NET "AD<0>"   LOC = "p5"   ;
NET "AD<1>"   LOC = "n6"   ;
NET "AD<2>"   LOC = "m6"   ;
NET "AD<3>"   LOC = "r6"   ;
NET "AD<4>"   LOC = "n7"   ;
NET "AD<5>"   LOC = "m7"   ;
NET "AD<6>"   LOC = "t7"   ;
NET "AD<7>"   LOC = "r7"   ;
NET "ALE"     LOC = "t5"   ;
NET "RDI"     LOC = "t4"   ;
NET "WRI"       LOC = "t3"   ;

NET "FPGA_INT6"   LOC = "N5"   ;
NET "FPGA_CS"     LOC = "P7"   ;
NET "DONE"        LOC = "B14"  ;

# FPGA<—>Flash  Interface
NET "Aout<0>"     LOC = "A5"   ;
NET "Aout<1>"     LOC = "A7"   ;
NET "Aout<2>"     LOC = "A3"   ;
NET "Aout<3>"     LOC = "D5"   ;
NET "Aout<4>"     LOC = "B4"   ;
NET "Aout<5>"     LOC = "A4"   ;
NET "Aout<6>"     LOC = "C5"   ;
NET "Aout<7>"     LOC = "B5"   ;
NET "Aout<8>"     LOC = "E6"   ;
NET "Aout<9>"     LOC = "D6"   ;
NET "Aout<10>"    LOC = "C6"   ;
NET "Aout<11>"    LOC = "B6"   ;
NET "Aout<12>"    LOC = "E7"   ;
NET "Aout<13>"    LOC = "D7"   ;
NET "Aout<14>"    LOC = "C7"   ;
NET "Aout<15>"    LOC = "B7"   ;
NET "Aout<16>"    LOC = "D8"   ;
NET "Aout<17>"    LOC = "C8"   ;
NET "Aout<18>"    LOC = "B8"   ;
NET "Aout<19>"    LOC = "C9"   ;
NET "Aout<20>"    LOC = "B10"  ;
NET "DQ<0>"       LOC = "A9"   ;
NET "DQ<1>"       LOC = "A12"  ;
NET "DQ<2>"       LOC = "C10"  ;
NET "DQ<3>"       LOC = "D12"  ;
NET "DQ<4>"       LOC = "A14"  ;
NET "DQ<5>"       LOC = "B14"  ;
NET "DQ<6>"       LOC = "A13"  ;
NET "DQ<7>"       LOC = "B13"  ;
NET "DQ<8>"       LOC = "B12"  ;
NET "DQ<9>"       LOC = "C12"  ;
NET "DQ<10>"      LOC = "D11"  ;
NET "DQ<11>"      LOC = "E11"  ;
NET "DQ<12>"      LOC = "B11"  ;
NET "DQ<13>"      LOC = "C11"  ;
NET "DQ<14>"      LOC = "D10"  ;
NET "DQ<15>"      LOC = "E10"  ;
NET "MEM_RESET"   LOC = "D15"  ;
NET "CE"          LOC = "E14"  ;
NET "OE"          LOC = "E15"  ;
NET "WE"          LOC = "E16"  ;
NET "WP"          LOC = "F12"  ;
```

```
# FPGA<->Radio Interface
NET "FPGA_IO0"     LOC ="P15"   ;
NET "FPGA_IO1"     LOC ="P14"   ;
NET "FPGA_IO2"     LOC ="N16"   ;
NET "FPGA_IO3"     LOC ="N15"   ;
NET "FPGA_IO4"     LOC ="M14"   ;
NET "FPGA_IO5"     LOC ="N14"   ;
NET "FPGA_IO6"     LOC ="M16"   ;
NET "FPGA_IO7"     LOC ="M15"   ;
NET "FPGA_IO8"     LOC ="L13"   ;
NET "FPGA_IO9"     LOC ="M13"   ;
NET "FPGA_IO10"    LOC ="L15"   ;
NET "FPGA_IO11"    LOC ="L14"   ;
NET "FPGA_IO12"    LOC ="K12"   ;
NET "FPGA_IO13"    LOC ="L12"   ;
NET "FPGA_IO14"    LOC ="K14"   ;
NET "FPGA_IO15"    LOC ="K13"   ;
NET "FPGA_IO16"    LOC ="J14"   ;
NET "FPGA_IO17"    LOC ="J13"   ;
NET "FPGA_IO18"    LOC ="J16"   ;

# FPGA <-> Sensor Interface
# AIO is connected to connector J2

NET "AIO_0"        LOC ="K1"   ; # J2, pin 5
NET "AIO_1"        LOC ="R1"   ; # J2, pin 7
NET "AIO_2"        LOC ="P1"   ; # J2, pin 9
NET "AIO_3"        LOC ="P2"   ; # J2, pin 11
NET "AIO_4"        LOC ="N3"   ; # J2, pin 15
NET "AIO_5"        LOC ="N2"   ; # J2, pin 17
NET "AIO_6"        LOC ="N1"   ; # J2, pin 19
NET "AIO_7"        LOC ="M4"   ; # J2, pin 21
NET "AIO_8"        LOC ="M3"   ; # J2, pin 25
NET "AIO_9"        LOC ="M2"   ; # J2, pin 27
NET "AIO_10"       LOC ="M1"   ; # J2, pin 29
NET "AIO_11"       LOC ="L5"   ; # J2, pin 31
NET "AIO_12"       LOC ="L4"   ; # J2, pin 32
NET "AIO_13"       LOC ="L3"   ; # J2, pin 30
NET "AIO_14"       LOC ="L2"   ; # J2, pin 28
NET "AIO_15"       LOC ="K5"   ; # J2, pin 26
NET "AIO_16"       LOC ="K4"   ; # J2, pin 22
NET "AIO_17"       LOC ="K3"   ; # J2, pin 20
NET "AIO_18"       LOC ="K2"   ; # J2, pin 18
NET "AIO_19"       LOC ="J4"   ; # J2, pin 16
NET "AIO_20"       LOC ="J3"   ; # J2, pin 12
NET "AIO_21"       LOC ="J2"   ; # J2, pin 10

# BIO is connected to connector J4
```

```
NET  "BIO_0"        LOC ="G2"   ;  #  J4 ,  pin  5
NET  "BIO_1"        LOC ="C1"   ;  #  J4 ,  pin  7
NET  "BIO_2"        LOC ="B1"   ;  #  J4 ,  pin  9
NET  "BIO_3"        LOC ="C2"   ;  #  J4 ,  pin  11
NET  "BIO_4"        LOC ="C3"   ;  #  J4 ,  pin  15
NET  "BIO_5"        LOC ="D1"   ;  #  J4 ,  pin  17
NET  "BIO_6"        LOC ="D2"   ;  #  J4 ,  pin  19
NET  "BIO_7"        LOC ="E3"   ;  #  J4 ,  pin  21
NET  "BIO_8"        LOC ="D3"   ;  #  J4 ,  pin  25
NET  "BIO_9"        LOC ="E1"   ;  #  J4 ,  pin  27
NET  "BIO_10"       LOC ="E2"   ;  #  J4 ,  pin  29
NET  "BIO_11"       LOC ="F4"   ;  #  J4 ,  pin  31
NET  "BIO_12"       LOC ="E4"   ;  #  J4 ,  pin  32
NET  "BIO_13"       LOC ="F2"   ;  #  J4 ,  pin  30
NET  "BIO_14"       LOC ="F3"   ;  #  J4 ,  pin  28
NET  "BIO_15"       LOC ="G5"   ;  #  J4 ,  pin  26
NET  "BIO_16"       LOC ="F5"   ;  #  J4 ,  pin  22
NET  "BIO_17"       LOC ="G3"   ;  #  J4 ,  pin  20
NET  "BIO_18"       LOC ="G4"   ;  #  J4 ,  pin  18
NET  "BIO_19"       LOC ="H3"   ;  #  J4 ,  pin  16
NET  "BIO_20"       LOC ="H4"   ;  #  J4 ,  pin  12
NET  "BIO_21"       LOC ="H1"   ;  #  J4 ,  pin  10
```

Appendix E

FPGA Makefile

```
## Xilinx fpga tool flow

TOPDIR   = ./
XDIR     := $(shell if [ -x "/usr/cad/Xilinx" ]; then echo "/usr/cad/Xilinx";\
               else if [ -x "/usr/local/Xilinx/" ]; then echo "/usr/local/Xilinx/";
SRC_DIR  = src
SOURCES  = $(SRC_DIR)/*
PROJECT  = mc8051_top
TOP      = mc8051_top

###########################################################################
# Xilinx tools
###########################################################################

XST_DEFAULT_OPT_MODE = Speed
XST_DEFAULT_OPT_LEVEL = 1
DEFAULT_ARCH = spartan3
DEFAULT_PART = xc3s400-ft256-4


XBIN = $(XDIR)/bin/lin
XENV = XILINX=$(XDIR) LD_LIBRARY_PATH=$(XBIN)

XST        = $(XENV) $(XBIN)/xst
NGDBUILD   = $(XENV) $(XBIN)/ngdbuild
MAP        = $(XENV) $(XBIN)/map
PAR        = $(XENV) $(XBIN)/par
BITGEN     = $(XENV) $(XBIN)/bitgen
PROMGEN    = $(XENV) $(XBIN)/promgen
FLOORPLAN  = $(XENV) $(XBIN)/floorplanner
IMPACT     = $(XENV) $(XBIN)/impact

XSTWORK    = $(PROJECT).work
```

```
XSTSCRIPT = $(PROJECT).xst

default:
    echo $(XSTWORK)
    echo $(SOURCES)

.PRECIOUS: %.ngc %.ngc %.ngd %.map.ncd %.bit %.par.ncd %cmd

ifndef XST_OPT_MODE
XST_OPT_MODE = $(XST_DEFAULT_OPT_MODE)
endif
ifndef XST_OPT_LEVEL
XST_OPT_LEVEL = $(XST_DEFAULT_OPT_LEVEL)
endif
ifndef ARCH
ARCH = $(DEFAULT_ARCH)
endif
ifndef PART
PART = $(DEFAULT_PART)
endif

$(XSTWORK): $(SOURCES)
    > $@
    for a in $(SOURCES); do echo "vhdl␣work␣$$a" >> $@; done

$(XSTSCRIPT): $(XSTWORK)
    > $@
    echo -n "run␣-ifn␣$(XSTWORK)␣-ifmt␣mixed␣-top␣$(TOP)␣-ofn␣$(PROJECT).ngc" >> $@
    echo "␣-ofmt␣NGC␣-p␣$(PART)␣-opt_mode␣$(XST_OPT_MODE)␣-opt_level␣$(XST_OPT_LEVEL)" >

# Synthesis step
%.ngc: $(XSTSCRIPT)
    $(XST) -ifn $<

# Requieres that $(PROJECT).ucf fails otherwise the rule will not match
%.ngd: %.ngc $(PROJECT).ucf
    $(NGDBUILD) -intstyle ise -dd _ngo -uc $(PROJECT).ucf -p $(PART) $*.ngc $*.ngd

%.map.ncd: %.ngd
    $(MAP) -o $@ $< $*.pcf

%.par.ncd: %.map.ncd
    $(PAR) -w -ol high $< $@ $*.pcf

%.bit: %.par.ncd
    $(BITGEN) -w -g UnusedPin:PullNone $< $@ $*.pcf

%.prm: %.bit
    $(PROMGEN) -o $@ -w -u 0  $<
```

```
%.mcs: %.bit
	$(PROMGEN) -o $@ -w -p mcs -u 0 $<

%.cmd: %.bit
	> $@
	echo "setMode -bs" >> $@
	echo "setCable -p parport0 " >> $@
	echo "adddevice -p 1 -part xcf02s -file $(PROJECT).mcs " >> $@
	echo "adddevice -p 2 -file $(PROJECT).bit " >> $@
	echo "program -e -p 2 " >> $@
	echo "exit " >> $@
#	echo "program -e -p 1 " >> $@
#	echo "identify " >> $@

program: $(PROJECT).cmd
	$(IMPACT) -batch $(PROJECT).cmd
impact:
	$(IMPACT) -batch clean.cmd

.PHONY: bclean
bclean:
	rm -fR _ngo xst
	rm -f *.work *.xst
	rm -f *.ngc *.ngd *.bld *.srp *.lso *.prj
	rm -f *.map.mrp *.map.ncd *.map.ngm *.mcs *.par.ncd *.par.pad
	rm -f *.pcf *.prm *.bgn *.drc
	rm -f *.par_pad.csv *.par_pad.txt *.par.par *.par.xpi *.par.unroutes
	rm -f *.bit
	rm -f *.cmd
	rm -f *.vcd *.vvp
	rm -f _impactbatch.log
```

Bibliography

[1] Bootstrap demo design - users manual. URL http://www.oregano.at/ip/mc8051/mc8051_bootstrap_ug.pdf.

[2] Atmel. Atmega128(l) complete. URL http://www.atmel.com.

[3] Martin Leopold, Mads Dydensborg, and Philippe Bonnet. Bluetooth and sensor networks: A reality check. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 103–113, November 2003. ISBN 1-58113-707-9. URL http://www.distlab.dk/public/distsys/publications.php?id=38.

[4] Kashif Virk. Testing fpga interfaces on hogthrob development platform. Technical Report Hogthrob-SoC-CSE-IMM-DTU-001, Informatics & Mathematical Modeling (IMM), Technical University of Denmark (DTU), 2004.

[5] Xilinx. Spartan-3 complete data sheet. Datasheet, 2004.