# The ADC66 Wireless ADC Board for the NRL-66 Machinery

Gergely V. Záruba
zaruba@uta.edu

Document Version: 2007-08-29

***Abstract –*** *This document describes the functionality, the hardware and software components of the NRL-66 ADC board that was developed during the author's summer research stay at the NRL Computational Multiphysics Laboratory, lead by Dr. John Michopoulos.*

## 1. Introduction

The NRL-66 ADC (short: ADC66) board has been developed to enable wireless reading of the six Wheatstone-bridge load cells of the NRL-66 machinery. However, it was built for a specific purpose, one of the design goals was to keep it generic and expandable, so it can be used for obtaining digital reading from other sources or even to provide output to digitally driven actuators. In general, all active components (and the bridge) are driven by 3.3VDC obtained either from the USB interface (USB option) or from an external power adapter or battery (wireless option).

### *1.1. ADC66 Architecture*

The general architecture of the ADC66 board is depicted in Figure 1, and consists of:

- A digital interface to the host machine, that is jumper selectable between an 802.15.4 compliant wireless-to-serial module (XBee) and a USB to serial module. The former option requires a wireless transceiver to be plugged into the host computer which is a secondary transceiver board that has been designed and built (TB66). The TB66 is also used when performing the initial programming of the XBee modules.
- A "master" microcontroller (short: 1611) is realized by TI's MSP430F1611 [10] running on an 8MHz crystal clock, interfacing serially (UART) to the digital interface module and providing an $I^2C$ interface to all the slave microcontrollers (see below). Port-1 of the 1611 is used to address the slaves individually through a (or several) demultiplexer (e.g., 74HC4514). This is needed to enable all the slaves to be programmed by the same code (and sending broadcast messages to them with their respective ports enabled to communicate their address to them). The 1611 has also eight LED-s (Port-2) attached for status displaying purposes.
- Slave microcontrollers (short: 2013) are used to obtain analog data in a digital form and do basic filtering on such data.
  Currently, the slaves are six MSP430F2013 [11] 16-bit sigma delta ADC microcontrollers with $I^2C$ support (but could be any $I^2C$ compliant microcontrollers). Each slave has an associated RJ45 connector that connects to one of the Wheatstone bridged load cells through a simple RC low-pass filter. The slaves have a single, shared JTAG connector for programming and need "jumper-magic" to be programmed.

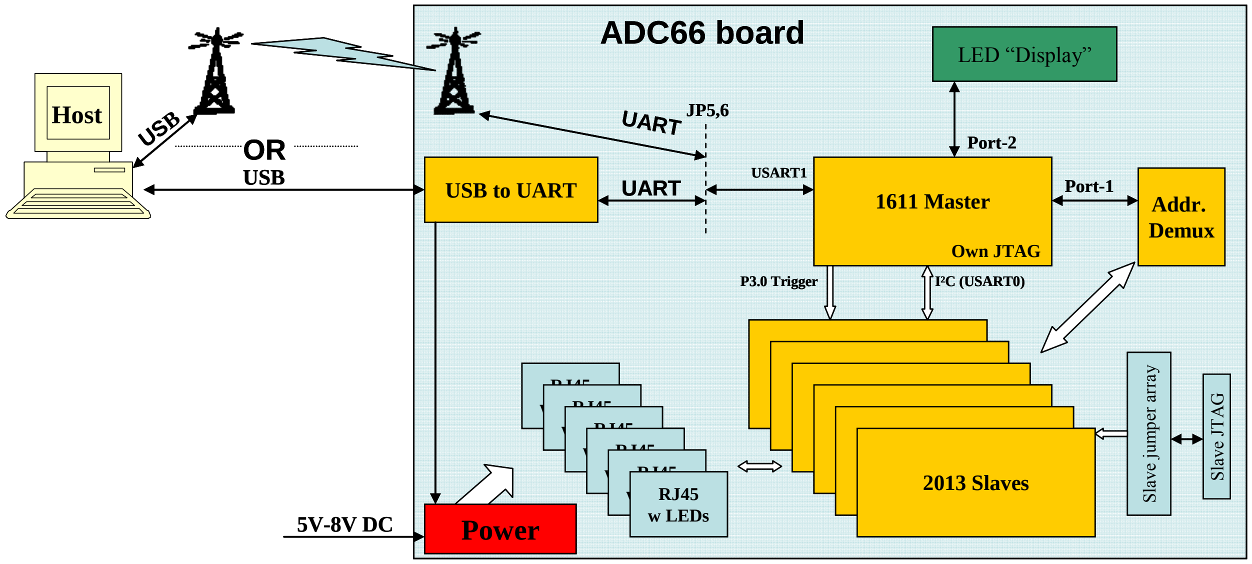


**Figure 1.** Overview of ADC66 Architecture

### *1.2. Choice of Microcontrollers*

The choice of the master and slave microcontrollers was made based on conversations with Dr. John Hermanson. The main reason for selecting the 1611 was the available Flash and RAM as well as the need for two USART interfaces. The reason behind selecting the 2013 as the slave was due to it 16-bit sigma-delta differential analog to digital input, its $I^2C$ capability and its form factor and price. The 2013 is essentially used as a programmable 16-bit ADC.

The FT232R IC from FTDI has been selected as the serial to USB converter for its wide driver support, price and performance. The XBee module has been selected for its versatility and price/performance.

## 2. Host – ADC66 Interface

The Host to ADC66 interface shows up as a serial port on the host regardless of the connection (wireless or the USB) option used; both of the connection options use the same serial to USB converter chip from FTDI. The drivers are readily available both on Widows and late-version Linux machines but if needed can be downloaded from [3]. The settings of the serial (COM) port are: **57.6k**baud, no software or hardware flow control, 8 bits, no start bit and 1 stop bit (**8-N-1**).

A "general" (7,4) Hamming code is used for communication between the host and the 1611. The (7,4) code encodes 4 bits into 7 bits and enables the correction of one. To transfer bytes, they need to be cut into lower 4-bits and higher 4-bits parts, each encoded with the (7,4) code. The "lower-4bits encoded into 7-bits" is transferred before the "higher 4-bits encoded into 7-bits". Since on the serial bus an 8 bit transmission is used and code words are only seven bits, we can create special code word sequences where the left-most bit is set one thus being able to use easy in-band signaling for frame-starts and -ends. Table 1 shows all code words and special sequences used. Appendix 1 has C code snippets implementing the (7,4) encoding, and error-correcting decoding. As an example, if we want to transmit a character "C" ASCII Hex: 0x43, we would need to transmit the following two codes: 0x1C 0x25 (corresponding to 3 {lower 4-bits} and 4 {upper 4-bits}).

Each frame transmitted between host and the ADC66 starts with the code word for START_FRAME and ends with the code word for END_FRAME. Frames from the Host to the

ADC66 are called ***commands*** while frames traveling in the other direction are called ***responses***. The very first byte assembled from the two code words following a START_FRAME describes the type of the command (command char) or response (response char).

**Table 1.** (7,4) codes and other code words used.

| Data | (7,4) code dec | (7,4) code hex | Data | (7,4) code dec | (7,4) code hex |
|---|---|---|---|---|---|
| 0 | 0 | 0x00 | 9 | 73 | 0x49 |
| 1 | 15 | 0x0F | 10 | 85 | 0x55 |
| 2 | 19 | 0x13 | 11 | 90 | 0x5A |
| 3 | 28 | 0x1C | 12 | 99 | 0x63 |
| 4 | 37 | 0x25 | 13 | 108 | 0x6C |
| 5 | 42 | 0x2A | 14 | 112 | 0x70 |
| 6 | 54 | 0x36 | 15 | 127 | 0x7F |
| 7 | 57 | 0x39 | **START_FRAME** | 129 | 0x81 |
| 8 | 70 | 0x46 | **END_FRAME** | 241 | 0xF1 |

Whenever the 1611 is reset, a short *comment-type* message is relayed to the host reporting its existence. During the compilation of the 1611 code a verbose mode can be turned on (by uncommenting the #define VERBOSE_MODE compiler directive in global.h and then remaking the code by make clean; make). If the 1611 works in a verbose mode, then most of the commands send to it will also result in human readable *comment-type* responses. For example a “Hello” command will respond with a “HELLO” comment in addition to a general response message.

The first command to the ADC66 should be a set-up command. Until such a command is received and the slaves are set up many of the other commands will fail.

## *2.1. Host Commands*

Each command issued by the host will generate a two byte response, where the first byte is whether or not the command succeeded and the second byte corresponds to the command. Command types are summarized in Table 2 (number of parameters are without escape characters and command/report character).

## *2.2. ADC66 Responses*

Response types are summarized and described in Table 3.

## *2.3. Examples*

**Example one:** checking whether ADC66 is alive

command issued:  H   (0x48)
transmitted frame:  0x81  0x46  0x25  0xF1
successful response:  OH  (0x4F  0x48)
corresponding response frame: 0x81  0x7F  0x25  0x46  0x25  0xF1

**Example two:** setting up 1 slave

command issued: S_  (0x53 0x01)

transmitted frame:  0x81  0x1C  0x2A 0x0F 0x00  0xF1
successful response:  OS  (0x4F  0x53)
corresponding response frame: 0x81  0x7F  0x25  0x1C  0x2A  0xF1

**Example three:** requesting reading from salve #5

command issued: A_  (0x41 0x05)
transmitted frame:  0x81  0x0F  0x25 0x2A 0x00  0xF1
successful response:  V_ _ _ _ _  (0x56  0x1F 0x05 0x32 0x8C B3)
corresponding response frame: 0x81  0x36 0x2A 0x7F 0x0F 0x2A 0x00 0x13 0x1C 0x63 0x46 0x1c 0x5A  0xF1
response's meaning:  reading sequence number: 31. ADC buffer of slave 5 is 35,890 which was assembled from 179 readings

**Table 2.** Command types.

| cmd. char | # params | Description |
|---|---|---|
| 'H' | 0 | ***Hello***; request an "alive" response. |
| 'S' | 1 | ***Set-up***; parameter is the number(!) of slaves to set up. This will force the 1611 to set-up the slaves, i.e., let the slaves know their address. It will generate "parameter" number of responses and comments. |
| 'A' | 1 | ***Get reading***; request the ADC buffer value of the slave pointed by the parameter (IDs start from 0) |
| 'P' | 2 | ***Periodic***; requests the 1611 to poll the (first parameter) slave periodically for conversion results and relay those results to the host. The second parameter is the period in 50ms increments. This is the default behavior. |
| 'R' | 2 | ***Set Rate***; will force the (first parameter) slave to set its running average learning rate to {(second parameter)/255}. The default value is 13, i.e., new samples have a weight of 0.05098. |
| 'T' | 0 | ***Trigger***; will ask the 1611 to generate a port-1 interrupt at the slaves, forcing them to reset their running average buffers. |
| 'N' | 2 | ***Number of Conversions***; will request the (first parameter) slave to restrict the number of samples averaged to the second parameter. It only is enforced if the mode of operation is "single". Default value is one. |
| '1' | 1 | ***Single Mode***; forces the (first parameter) slave into single mode, i.e., after taking a "parameter" amount of samples the respective 2013 turn off its ADC. |
| '9' | 1 | ***Continuous Mode***: forces the (first parameter) slave into continuous mode, i.e., the running average is continuously updated. This is the default mode. |
| '0' | 1 | ***Soft Reset***: resets the number of samples (thus the ADC buffer) of the slave pointed to by the first parameter. This may be used as an asynchronous trigger. |

**Table 3.** Response types.

| resp. char | # bytes | Description |
|---|---|---|
| ‘C’ | many | ***Comment***; the rest of the frame contains ASCII characters with “human understandable” information. |
| ‘V’ | 5 | ***Values***; the first byte returned is a sequence number running from 0 to 127. The second byte is the ID of the slave. The third and fourth bytes are the lower and upper byte part of the 16-bit ADC buffer respectively. The fifth byte is the number of samples used in averaging the ADC buffer.<br><br>If periodic reporting is enabled these messages can be generated asynchronously (i.e., without a preceding command). |
| ‘E’ | 1 | ***Error***; when receiving a command, and encountering an error in the command (e.g., wrong command length, decoding error), this message will be relayed with the original command char following ‘E’.<br><br>If there was a general error (e.g., not even the command character was understood), then a character ‘G’ (for general) is sent back. |
| ‘O’ | 1 | ***OK***; behaves similarly to the Error response except it signals an acknowledgment to a correctly received command. The exception is the “Get Reading” command for which there is no OK response. |

## 3. Master – Slave Internal Interfaces

There are three interfaces between the 1611 master and the 2013 slaves:

1. The data interface between the 1611 and the 2013-s is an $I^2C$ [6] physical interface for which the 1611 has native register support and the 2013 can be programmed to support.
2. The trigger interface is a single common port (P3.0 on the 1611 and P1.4 on the 2013-s) that the 1611 can use to force conversion at the slaves. This means that if P3.0 is raised an interrupt will be generated in all slaves, forcing them to reset their buffers and restart their ADC-s.
3. The address interface is Port-1 of the 1611 demultiplexed to individual 2013-s on their P1.2 input. P1.7 of the 1611 is used to disable the demultiplexing, thus a total of 126 slaves are possible (address 0 is the broadcast address, and 127 is the master). This feature enables that all 2013-s be programmed with the same code; the 1611 will send a broadcast $I^2C$ message to a port-1 addressed slave to set its $I^2C$ address. For example if Port-1 of 2013 outputs 0x03 it means that slave #4 (as salves are numbered from zero but their addresses start from 1) is going to receive an $I^2C$ broadcast communication telling it that it indeed is slave #4. To disable demultiplexing the 1611 should keep the MSB of Port-1 high.

### 3.1. *I$^2$C Communication*

Due to the lack of memory in the 2013 there is no error detection or correction on the I$^2$C bus. However, the clock generation and the acknowledgment bits are somewhat countering the need for such error detection/correction scheme.

The I$^2$C standard divides modules on the bus by their function into four categories: *master transmitter* (clock is generated, slave is addressed, slave is written, acks are received), *master receive* (clock is generated, slave is addressed, slave is read, acks are transmitted), *slave receiver* (clock is received, address is received, data is received, acks are sent), and *slave transmitter* (clock is received, address is received, data is transmitted, acks are received). Every eight bits of data must be followed by an acknowledgment bit (pulling SDA low) from the receiver. The first bit of the communication describes whether the master is in receive or transmit mode while the remaining seven bits designate the address of the slave addressed.

In the ADC66, the only *I$^2$C master* on the bus is the 1611, i.e., none of the 2013-s can generate the clock or initiate data transfer. The 1611 can work both as an *I$^2$C master transmitter* in which case it will send commands to a 2013 slave or as an *I$^2$C master receiver*. Regular commands transmitted from the 1611 when it is in a *master transmitter* state are summarized in Table 4. When the 1611 is in a *master receiver* state, it implicitly requests the addressed slave to report its ADC buffer and buffer size. The report size (when the 1611 is in a *master receiver* mode) is always three bytes long. The first byte is the low-byte of the ADC buffer, the second byte is the high-byte while the third byte corresponds to the buffer size.

When the 1611 is in a *master transmitter* state and sends a broadcast message, it will always enable one of the slaves' P1.2 ports by its own corresponding Port-1. In this case the payload is always one byte long and carries the future I$^2$C address of the P1.2 enabled slave. One technicality to remark here upon is that slave numbers are internally increased by one, i.e., slave#0 will have address: 1, slave#1 will have address: 2, etc.

## 4. Hardware

This section outlines the hardware design, and hardware features of both the ADC66 board and the TB66 transceiver module.

### 4.1. *ADC66 Hardware*

The main architecture of the ADC66 was given in Figure 1. In this section we provide pictures of the real hardware, describe jumper settings, and programming procedures of both the 1611 and the 2013-s. A schematic for the hardware can be found in Appendix 2 (made partially with Kicad), while Appendix 3 shows locations of various components on the board. Figure 2 depicts the ADC66 board for our "aesthetic pleasure". Figure 3 shows the ADC66 board with various jumpers, connectors, and LEDs identified (with reference to the schematics).

#### *4.1.1. DC*

The DC power jack is used to supply the ADC66 with power if not operating from power drawn from the USB connector. The DC plug can take voltages from 5VDC to 8VDC with the positive terminal located in the middle of the jack.

#### *4.1.2. USB*

The USB connector is a mini USB 2.0 socket. It can be used to talk to the ADC66 (if respective jumpers are set) and/or to provide power to the ADC66.

**Table 4.** $I^2C$ commands.

| cmd. char | #params | Description |
|---|---|---|
| 0x01 | 1 | ***Set Conversion***; the parameter is the number of conversions to use in the running average ADC buffer if operating in a "single" mode. |
| 0x02 | 1 | ***Set Rate***; the parameter is a function of the learning rate of the running average buffer: {parameter}/255 |
| 0x04 | 0 | ***Set Continuous***; set the slave to continuous operation mode. |
| 0x08 | 0 | ***Set Single***; set the slave to single operation mode. |
| 0x10 | 0 | ***Reset***; reset the slaves ADC buffer and counter. |

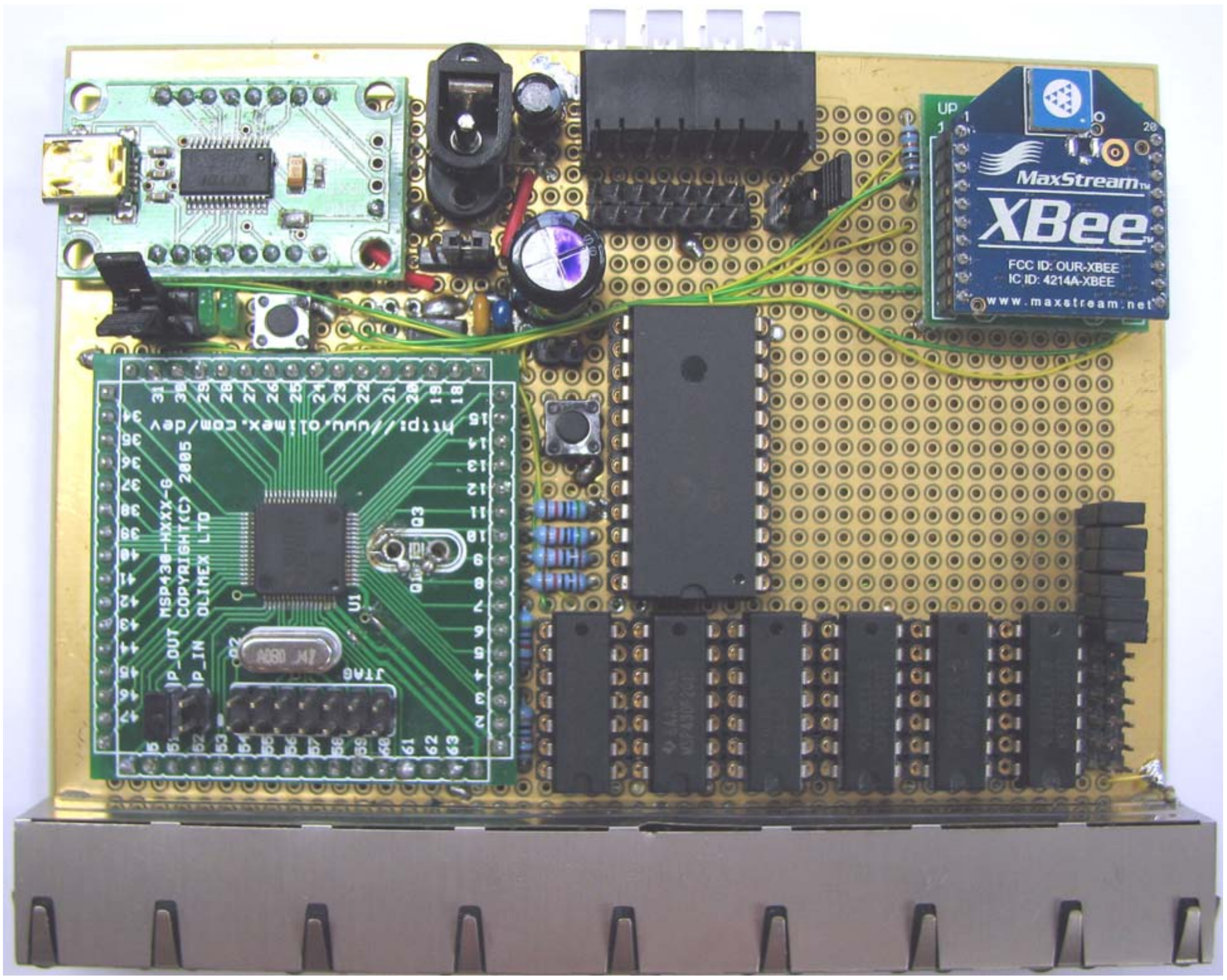


**Figure 2.** ADC66 Board (unedited).

### *4.1.3. JPP*

The JPP jumper selects the power source for the ADC66. If it is jumpered on "1-2" (as shown in the figure), the DC power jack is selected as the power source. If it is jumpered "1-3", then the USB interface provides the power. A word of caution: the current drawn from the USB connector can be in the order of 0.5A. Even if the ADC66 is set-up to use USB based communication. the JPP may be plugged to "1-2" powering the ADC66 from the DC plug.

### *4.1.4. JPR and JPTT*

JPR and JPT determine the method of communication with the host; they should always be set together. If set to “1-2” (as shown in the figure) wireless communication (the XBee module) is selected. If they are set to “1-3” the USB communication method is selected.

### *4.1.5. RXLED and TXLED*

If USB communication is enabled, the RXLED and TXLED will flash whenever data is received or transmitted on the USB interface.

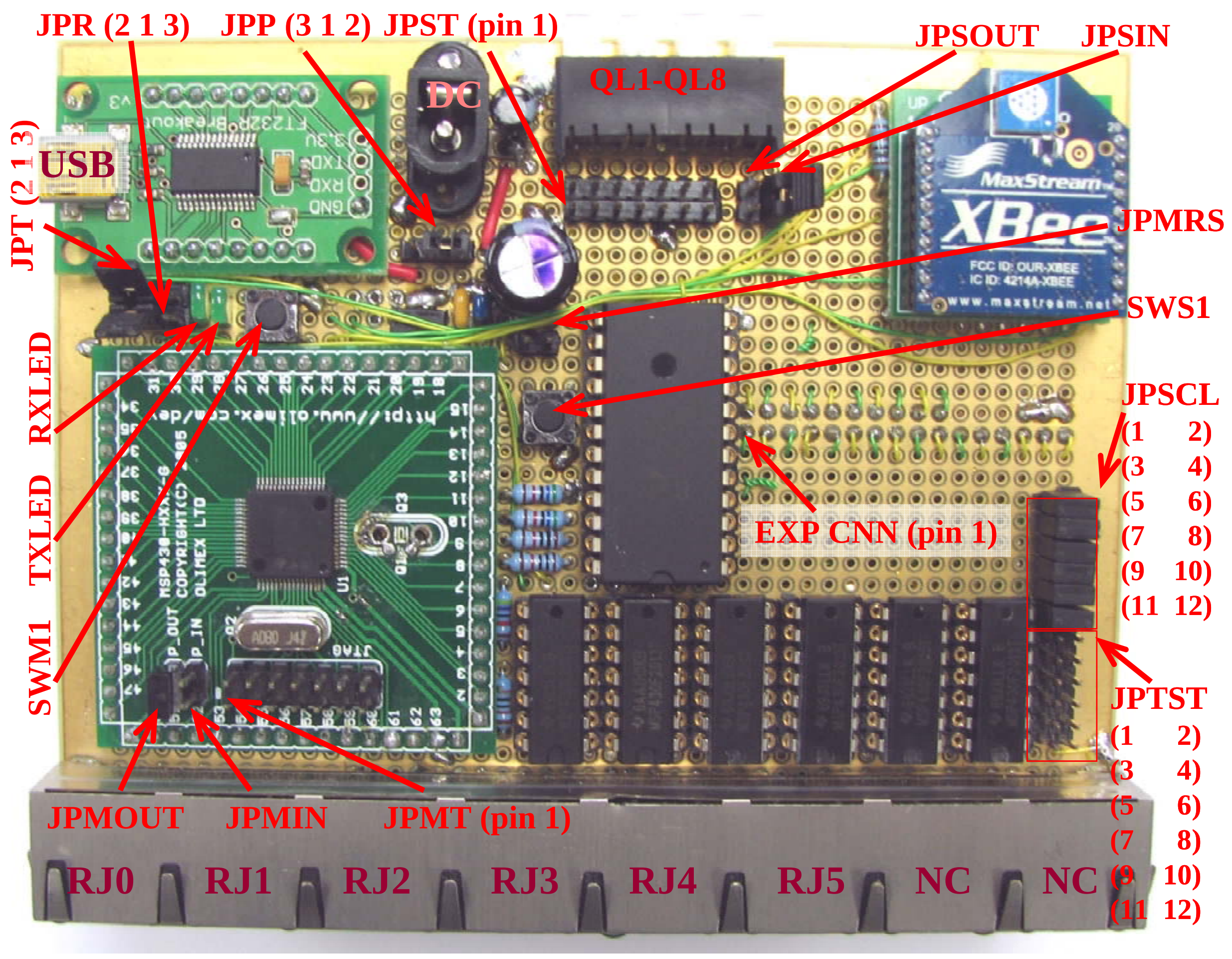


**Figure 3.** Jumpers, connectors, and LEDs on ADC66.

### *4.1.6. QL1-QL8*

These are general purpose LEDs driven from Port-2 of the 1611. They can signal the status of the system or help with debugging. Currently QL1 is set to flash periodically, indicating that the module is operational.

### *4.1.7. RJ0-RJ5*

RJ0 to RJ5 are RJ45 receptacles to connect to the measured Wheatstone bridges. The pin-out is shown in the schematics in Appendix 2. Each RJ45 connector has two LEDs built into its case, driven by the corresponding 2013-s (port-2). The yellow LED (Y-LED) is flashing very quickly (seems to be on) when the corresponding 2013’s ADC is operational. The green LED will flash slowly (about once in a second) if the 2013 is powered and will flash more rapidly (about twice a second) if the corresponding 2013 has received its address.

### *4.1.8. SWM1 and JPMRS*

The SWM1 switch and the JPMRS jumper are used to reset the 1611. The jumper can be used to keep the 1611 switched off (in a reset state), a requirement when programming the slaves.

### *4.1.9. SWS1*

SWS1 may be used to reset all 2013s, as their respective reset pins are connected together.

### *4.1.10. EXP CNN*

The expansion connector can be used to connect daughter boards to the ADC66 motherboard. Such a daughterboard is detailed in Appendix 5. The pin-out of the expansion connector is given in Table 5.

**Table 5** Expansion connector pin-out.

| Pin | Function | Pin | Function | Pin | Function | Pin | Function |
|---|---|---|---|---|---|---|---|
| 1 | SBUS-6 | 2 | TEST [11] | 3 | SBUS-7 | 4 | SCL/P1.6 [8] |
| 5 | SBUS-8 | 6 | SCLK/P1.5 [7] | 7 | SBUS-9 | 8 | SMCLK/P1.4 [6] |
| 9 | SBUS-10 | 10 | SDA/P1.7 [9] | 11 | SBUS-11 | 12 | RESET [10] |
| 13 | SBUS-12 | 14 | P1-BUS P1.0 | 15 | SBUS-13 | 16 | P1-BUS P1.1 |
| 17 | SBUS-14 | 18 | P1-BUS P1.2 | 19 | SBUS-15 | 20 | P1-BUS P1.3 |
| 21 | N/C | 22 | P1-BUS P1.4 | 23 | N/C | 24 | P1-BUS P1.5 |
| 25 | N/C | 26 | P1-BUS P1.6 | 27 | N/C | 28 | P1-BUS P1.7 |
| 29 | GND | 30 | +3.3V | 31 | GND | 32 | +3.3V |

### *4.1.11. Programming the 1611 Master*

The code for the 1611 has been written to be compiled with mspgcc [7]. We have used an OLIMEX made USB JTAG programmer [8] (with an OLIMEX made MSP430F161 breakout board [9]). The general 14pin JTAG programming connector is JPMT. The JPMIN and JPMOUT jumpers are mutual, only one and exactly one of them needs to be jumpered during programming. If JPMOUT is used, then power to the ADC66 has to be supplied through either the USB or the DC jack (see their discussion above). If JPMIN is used then all(!) external power needs to be disconnected and it is recommended that all RJ connectors are disconnected as well (so they do not draw power from the programming unit). After programming, it is recommended to apply a reset. We have mostly used the JPMOUT option.

### *4.1.12. Programming the 2013 Slaves*

Programming the slaves is a little more involved as they are connected together and share a JTAG connector, yet only one unit a time can be programmed. As the 2013-s have their pins multiplexed (i.e., among other with functions of the $I^2C$) the 1611 has to be held in reset state during programming (e.g., by placing a jumper on JPMRS or holding SWM1 down).

The JTAG connector of the 2013-s is JPST, with JPSIN and JPSOUT as the power selectors for programming (see description of JPMIN and JPMOUT in the previous section). It is recommended that the JPMIN option is used with all(!) other connectors (including RJ45 connectors) to the ADC66 disconnected. We have used a TI MSP430-FET430UIF debugger programmer [13] with the IAR Kickstart [12] suite to program the 2013-s (as neither gcc nor the OLIMEX programmer do fully support the MSP430F2013).

**Figure 3 shows the jumper settings of JPSCL and JPTST when the board is not being programmed**, i.e., all jumpers on JPSCL need to be on ( “1-2”, “3-4”, “5-6”, “7-8”, “ 9-10”, "11-12") and no jumper on JPTST need to be placed. This is needed to connect all 2013 I$^2$C SCL pins together that need to be separated for programming.

To program a slave some of its pins need to be connected to JPST (e.g., TEST) while some pins of other slaves need to be disconnected (i.e., P1.6 I$^2$C SCL). After the JTAG connector has been placed on the JPST and the 1611 has been disabled, we need to select the slave to be programmed. This is accomplished by jumpering JPSCL and JPTST appropriately. To program slaves exactly one jumper can be on JPSCL and exactly one jumper can be on JPTST at corresponding positions, e.g., "1-2" on both of them correspond to slave number zero (closest to the 1611), "3-4" on both of them correspond to slave number one, "5-6" on both of them correspond to slave number two, "7-8" on both of them correspond to slave number three, "9-10" on both of them correspond to slave number four, "11-12" on both of them correspond to slave number five. Figure 4 shows jumper setting for programming slave number one (note the jumper on JPMRS as well).

A word of advice: in order to be able to program the 2013 slaves this way, one has to make sure that none of them use their JTAG related pins (P1.4, P1.5, P1.6, P1.7) as outputs. A new 2013 may need to be programmed separately (out of the ADC66) first if it had a non-compliant code on it before.

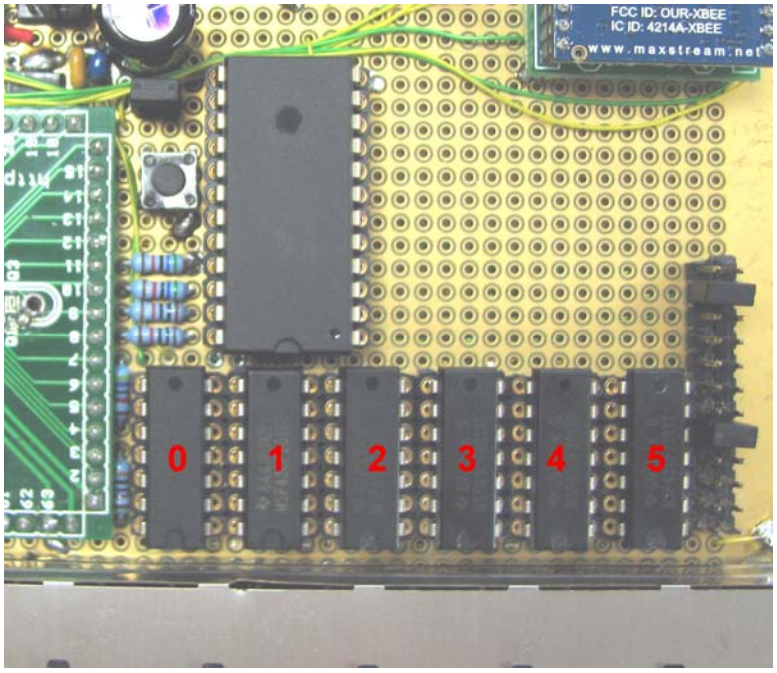


**Figure 4.** Programming slave number one.

### *4.1.13. XBee Transceiver Module*

A new XBee module needs to be programmed before placing it into the ADC66. This programming can be done using the TB66 board with the instructions provided in Section 4.2.1.

### *4.1.14. Power Requirements*

All components of the circuitry have been chosen to operate at 3.3VDC. Although the TI microcontrollers are low-power devices they function in a power-hungry mode (constantly converting analog signals). In addition, the power supply has to deal with the requirements of the XBee module (~50mA), the excitation of the bridges (~10mA / each), and driving LEDs (~10mA/each). To enable a power supply as low as 4.5V, a low dropout voltage (LDO) regulator needs to be chosen. Unfortunately, as a rule of thumb, the more current a regulator can withstand, the higher its dropout voltage. Currently, the regulator employed (AME8815AEGT330Z) [1] can supply as much as 1.5A with a dropout voltage of around 0.6V, thus an operation from 4V may be possible. The regulator is protected by a schottky-diode to ensure that its input never goes way below its output (e.g., due to transients when unplugging power). The input and output of the regulator are buffered by elcaps (electrolytic capacitors) as well as ceramic capacitors (due to the elcaps' high frequency behavior).

### *4.1.15. Bridge Excitement*

Currently the ADC66 uses the same 3.3V power to excite the Wheatstone-bridges on its RJ45 outputs as it is using for all other circuitry. As load cells have a typical 3mV/V resolution at maximum physical load (i.e., if fully loaded they will output about 9.9mV at 3.3V excitation), the precision of their output greatly depends on the stability of the excitation. It is recommended that they have their own rail of power (even if it is from the same regulator – as in the schematic) with high-quality capacitive buffering.

To make readings more precise, a higher excitation voltage may be chosen. The rule of thumb is to have the average excitation (between positive and negative excitation) around 1.65V $\{(V^-+V^+)/2=1.65\}$. Thus, at a 20V excitation (where the max output on the bridge may be as high as 60mV) the excitation potentials need to be: $V^-$=-8.35V and $V^+$=11.65V. High-current, variable charge pump DC-DC converters such as the MAX629 [4] may be used in such applications.

### *4.1.16. Analog Filtering*

The 2013-s provide a differential input for their AD conversion. In order to filter out unwanted components of the Wheatstone-bridge signal, and to limit the signal's bandwidth, a low pass filter is required. Currently the ADC has a simple one-stage R-C low-pass filter with a -3dB cutoff at around 1.5kHz (RSF=10kΩ CSF=100nF) on all Wheatstone inputs. It may help the precision if CSF-s were replaced by 1μF or higher.

It is recommended that filtering components be small SMDs, and that wires from RJ45 inputs to the ADC inputs of the 2013-s are either protected by ground wires and planes around them, by shielded cables, or by twisted pairs (currently).

### *4.1.17. Expanding the System*

The ADC66 has been designed to be expandable by more slaves. The expansion connector should have all necessary signals for and expansion. Additional boards may be attached; currently the USA demultiplexer has 10 more available ports (see SBUS-OUT), but more demultiplexers may be added; the number of slaves can be as much as 126. Details on such a daughterboard can be found in Appendix 5.

If attaching more 2013 slaves, they will need to be connected the same way as existing ones are, i.e., their pins: 1,6,7,9,10,14 connected to the corresponding pins of the other slaves, and their pins 11 and 8 connected appropriately to a (new) block of JPTST-s and JPSCL-s.

If actuators need to be driven by digital signals, a slave microcontroller needs to be chosen so as to have two serial ports (e.g., the MSP430F1611) one to be a slave on the $I^2C$ bus and another to talk to the actuators.

### *4.1.18. Some Hardware Design Issues*

The prototype board has been assembled using breakout boards, legged and SMD passive components, and PDIP ICs (if available). Wire-wrap wires were soldered and used as rails between components.

The $I^2C$ bus operates at around 50kHz. We have found that glitches on the bus (due to hardware problems with the microcontrollers) are less likely if the $I^2C$ clock is asymmetric biased significantly toward a high-state.

The 1611 is recommended to have an external crystal for more reliable UART communication. Currently an 8MHz crystal is used (and the firmware is written accordingly) that requires 18pF of load capacitance. Since the 1611 requires loads on both XIN and XOUT pins and these capacitors are viewed as series-connected from the viewpoint of the crystal, (and there is a ~3pF nominal capacitance of the pins themselves), 33pF loads are used on both pins. As the breakout board did not allow for a reliable operation of a watch crystal on XT1, this functionality has been removed (the board's primary purpose is not to have very-low power consumption anyway). We recommend the reader to follow up on TI released errata-s to see the many bugs and workarounds with these microcontrollers. The 1611 has many unused ports, free cycles, and enough memory to support its own AD conversions.

The 2013s are operating from internal DCO clocks that have a relatively large (but for our application insignificant) deviation among components. They are set to run at around 20MHz to avoid timing and glitch problems on their $I^2C$ ports. It is speculated that by running them all from an external 20MHz clock source, their ADC performance would go up, however such design would have to deal with the EMF of that clock source as well as would have to miss out on one of the LEDs for display (unless 2-wire Spy-by-wire would be used for programming instead of the 4-wire JTAG). At this clock with the current oversampling settings and clock dividers, the 2013s can do about 600 conversions a second.

## *4.2. TB66 Transceiver Board Hardware*

The transceiver board (TB66) needs to be connected by USB to the host if the ADC66 is operating in a wireless mode. The TB66 uses the same USB to serial converter as the ADC66 uses so no additional drivers need to be installed; in fact the host computer will not notice a difference in communication. The TB66 is depicted in Figure 5 and its schematic can be found in Appendix 4.

The TB66 is a very simple board containing three major parts: the USB to serial converter to talk to the radio (XBee) module, the XBee module, and a power regulator for the XBee board to be supplied from the USB (as the 3.3Vout option of the USB to serial converter cannot supply enough current for the XBee to operate). The TB66 may also be used for initial programming of XBee modules from a host and thus should have appropriate (2mm spacing) sockets so XBee modules can be replaced. There is a reset button to reset the XBee module if needed.

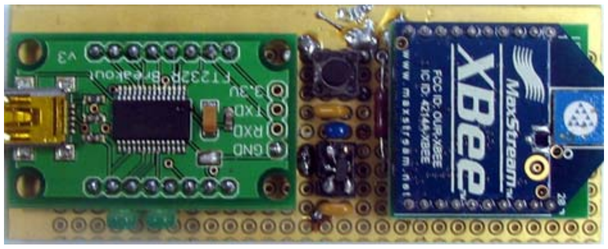


**Figure 5.** The TBA66 board.

### *4.2.1. XBee Transceiver Module*

The XBee transceiver module is manufactured by MaxStream Wireless [5]. It provides a programmable, off-the-shelf serial to 2.4GHz wireless (802.15.4 compliant) interface. When the module is shipped it is configured to default factory settings which need to be changed for it to operate in the TB66 and ADC66. Detailed instructions to the XBee module can be found in [5].

The default settings of the XBee module are to operate at 9.6kbaud with 8-N-1. If a new XBee module needs to be programmed, it should be placed into the TB66 board, and then the TB66 needs to be connected to a host. A Hyperterminal application (on Windows or similar application on Linux) can be used to program the XBee module, by opening the appropriate "COM" port (the one that is recognized as a USB to serial port by the operating system) with the above serial communication settings.

To enter XBee's programming mode, three "+" characters have to be sent to it (e.g., by pressing "+" three times after each other quickly (make sure all other surrounding XBee modules are turned off). The XBee module will respond with an "OK" message signaling that it has indeed entered programming mode. Next, the following commands have to be entered relatively quickly so the module does not time out:

```
ATCH0C <CR>    (setting the wireless channel of operation)
ATPL4  <CR>    (set power level to maximum - 0dBm)
ATSM0  <CR>    (turn off sleep mode)
ATRO50 <CR>    (set inter-char timing - to avoid retransmission problems)
ATBD6  <CR>    (set baud rate to 57.6k)
```

After the last command, the Hyperterminal will loose connection to the module. It will need to be reestablished by restarting Hyperterminal and setting the baud rate to 57.6kbauds. After setting up a connection to the module again we need to enter the program mode again (three "+"-s) and then enter the following commands:

```
ATWR   <CR>    (write configuration to flash memory)
ATCN   <CR>    (exit command mode)
```

### *4.2.2. Power*

A low drop-out voltage (LDO) regulator needs to be used to power the XBee module (3.3V) from the USB (5V) that can supply at least 50mA continuously. In our design we are using a FAN2504S33X [2] SMD regulator. It is not recommended to use the USB to serial module's 3.3V supply feature to power the XBee module.

## 5. Firmware

The firmware for both the 1611 and the 2013-s can be downloaded from [16]. The mspgcc [7] compiler suite and gnu make utilities have been used to develop the code for the 1611, while IAR Kickstart from TI [12] has been used to develop code for the 2013 (due to lack of proper support from mspgcc). There are slight differences between these compilers and how they handle direct memory writes, interrupt functions and interrupt handlers. The 2013 code contains macros to enable its compilation with the mspgcc compiler; however such attempt will run out of memory. As the free version of the Kickstart compiler has a memory limit, the 1611 code has been developed with mspgcc. For microcontroller programming instructions see Sections 4.1.11 and 4.1.12.

## 5.1. The 1611 Firmware

The 1611 firmware is spread over 15 files (1 Makefile, 7 header files, and 7 code files). The code can be compiled by issuing a "*make*" command. A "*make clean*" command will remove most things unnecessary from the build directory and should be used whenever modifying a header file, before issuing "*make*". If using the OLIMEX USB programmer (and assuming that the programmer files [8] are in the search path), a "*make download*" command will program the 1611 device. There has been a considerable effort to make the code readable and commented.

### 5.1.1. The "global" files

The *global* files contain all global variables (and their definitions in the header file), useful macros, and global setting macros for the firmware. It also includes general code, such as a pause routine and functions to decode the (7,4) Hamming code. *global.h* has a VERBOSE_RESPONSE macro that can be uncommented to relay human readable responses with each communication (currently uncommented).

### 5.1.2. The "init" files

The *init* files contain all peripheral initialization code for the 1611. These include: disabling the low-power detection watchdog; setting up the clock to use the crystal; initializing the software clock with Timer-A (crystal clock divided by 8) mainly used for giving a 50ms heart rate; setting up USART0 as an $I^2C$ master, and USART1 as a UART communication peripheral at 57.6kbaud, 8-N-1.

1611 ports are also set up here. Port-1 is used to address slaves (thus they are all general digital outputs). Port-2 is all general digital outputs to drive LEDs. Port-3 has the P3.0 trigger for the slaves as well as the USART0 and USART1 peripherals. Port-4 connects to the XBee module for possible control (unused). Ports 5 and 6 are unused.

### 5.1.3. The "main" files

The *main* files contain the main execution loop of the firmware. Execution will only exit this thread of execution when interrupts are being handled. Interrupts interact very heavily (in a quasi-parallel) manner with the main loop by sharing information over global variables. The main loop essentially waits for two interrupt handlers to notify it (through global indicator variables) that "something new has happened". These two main parts are:

- the handling of the heart-rate, i.e., about every 50ms the main loop (forced by a Timer-A interrupt interaction) is going to enter an execution thread where output LEDs are flashed (if enabled). In addition, if there is a periodic slave request, data is polled from the appropriate slave through $I^2C$ (synchronous) and is relayed to the host over the UART (asynchronous).
- if a new character has been received over the UART from the host (as indicated by a global variable set by incoming serial interrupts), an execution thread is entered that tries to parse the incoming serial buffer for a command (a set of characters between START_FRAME and END_FRAME). If such a command is found then the command is decoded, the command byte is extracted, the number of parameters is checked, a response is relayed back to the host, and a corresponding code is executed.

### 5.1.4. The "interrupts" files

The *interrupts* files contain all interrupt handlers, i.e., handlers that are automatically invoked: when a character is received from UART; when the transmission of a character over USART has been finished; if an event has occurred over the $I^2C$ bus; if Timer-A has reached a count up corresponding to fifty milliseconds (50,000 divided clock hits); or when Timer-A overflows. More about the behavior of these peripherals is in their respective files.

### 5.1.5. The "serialin" files

The *serialin* files contain code that is invoked by a UART character reception interrupt. The *serialin* code manages a circular buffer (with a head and a tail pointer), in which data is placed. It is the main loop's responsibility to make sure that this buffer is periodically read so as to not cause an overflow. The *serialin* files contain functions for handling the incoming serial buffer for the main loop to read, peek into, or delete entries. The circular behavior is hidden from all other functions (it is somewhat like a C++ object).

### 5.1.6. The "serialout" files

The *serialout* files contain code that manages the output buffer for the UART communication with the host. Just like *serialin*, *serialout* is using a circular buffer in which other code can insert characters without knowing about its circular structure. *serialout* is also in charge of encoding all communications with the (7,4) Huffman code, and has capabilities to easily insert "OK" and "Error" messages in the buffer. UART serial transmission is asynchronous (non-blocking), i.e., threads that are writing to the buffer do not have to (and cannot) wait for the data to be transmitted. The interrupt handler responsible for UART transmissions will interact with the transmission buffer (mutuality is, and has to be managed) and if the buffer is not empty, then transmit the next character.

### 5.1.7. The "i2c" files

The $I^2C$ files together with the appropriate interrupt handler are responsible for managing the master-receiver and master-transmitter operation of the 1611. The $I^2C$ transmitting and receiving functions are synchronous (blocking), i.e., when calling them the execution has to wait until bytes are sent or received. To transmit or receive any information (before calling the write or read functions), global variables have to be set, filling out the number of bytes to be transmitted/received and the buffer to be transmitted. The write and read functions will block until the state automaton implemented in the $I^2C$ interrupt handlers will release a global variable signaling either an error condition or a success.

## 5.2. 2013 Firmware

The 2013 firmware is relatively simple and is contained in a single file. Currently the code is exhausting all available memory in the 2013. The code is extensively commented.

### 5.2.1. The main function

First the watchdog (low power reset) is disabled and then the temperature coefficient of the SDA16 is fixed (see device errata [14]. The DCO clock is set up to operate at its highest frequency – around 20MHz, and to source the master clock MCL and SMCL. Timer A is set up to count up with SMCL to 0xFFFF and generate an interrupt when overflowing; this is used to enable LED flashing.

Port-1 is set up to output $V_{REF}$ on P1.3 (as recommended by TI) while raising edge trigger interrupt is enabled for P1.4. Port-2 is used to drive LEDs (as it is not used for driving and reading an external crystal/clock).

The USI (programmable serial interface) is set up for $I^2C$ support in slave mode. Next, the SDA16 is set up to continuously sample on A0 (P1.0 and P1.1) using a built-in 1.2V reference, a gain of 32 and oversampling 1024 times (see SDA16 documentation for details [15]). Finally, interrupts are enabled.

In the main loop, first the green LED flashing is taken care of. Depending on whether the slave address has been set, the countdown variable *i* will be set, and when zero, the green LED's status will be changed (XOR-d). Then we make sure that the ADC is running if we are operating in a continuous mode or if the number of samples requested is less than the number collected. Finally, we deal with any new SDA16 obtained result, by adding it to our running average ADC buffer (and flash the yellow LED). Floating point operations (especially mixing them with integers) should be avoided as much as possible.

### *5.2.2. The USI ($I^2C$) Interrupt Handler*

Built in $I^2C$ support is somewhat limited in the 2013. This interrupt vector will be called whenever we have transmitted a given amount (USICNT) of bits. To deal with $I^2C$ in a proper manner, a state automaton has to be set up. It may appear strange that we are "sending" NACK messages although there is no such thing in $I^2C$ (keeping the SDA high is not acknowledging anyway); however this is done to maintain the states of our automaton. The interrupt handler is also responsible of receiving and interpreting commands as well as assembling a response when the 1611 is requesting data.

### *5.2.3. Timer-A Interrupt Handler*

Timer-A is used to enable coarse timing mainly used for flashing the green LEDs. It is set to increment to 0xFFFF after which an overflow occurs, calling this interrupt handler routine. The handler will set a global variable true (which will be reset in the main loop of main after it has been processed).

### *5.2.4. SDA16 Interrupt Handler*

This interrupt handler will be called when a new SDA16 result is available. It will copy this new result in a global variable (and set another global variable) to be processed in the main loop. After turning on the continuous sampling the first three samples will be thrown away by the 2013 hardware.

### *5.2.5. Port-1 Interrupt Handler*

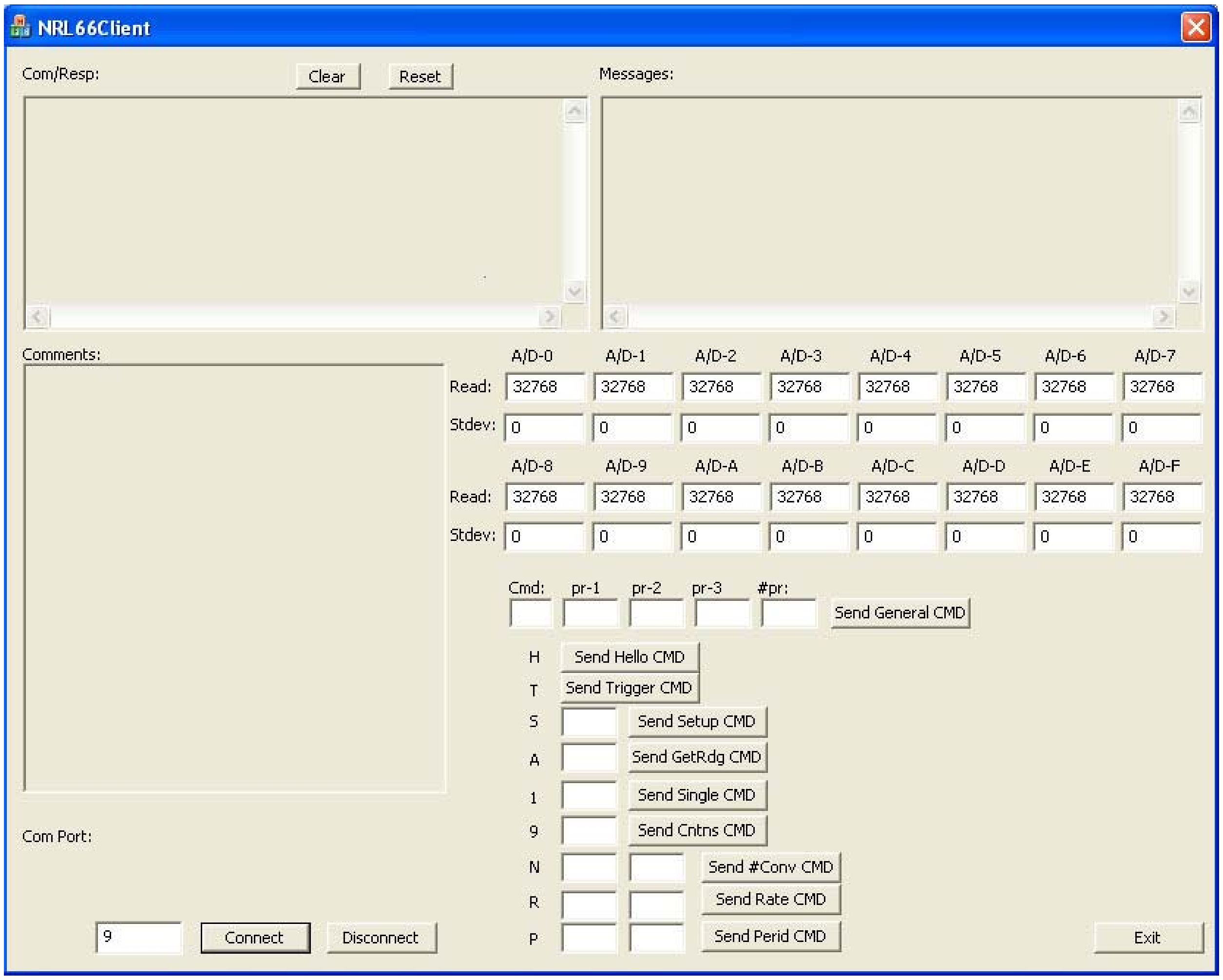
This interrupt handler is called when Port1.4 of the 2013 is raised (by Port3.0 of the 1611) thus triggering the 2013 to restart its sampling process. The SD16 will be restarted and the number of samples set to zero (which will force the running average buffer to be emptied).

## 6. Simple Visual-C++ Client

A simple Visual-C++ written client is provided [16] in both a compiled and source code form to test the ADC66. No programming documentation is made available for this client software; however its operation is simple (using separate threads for the interface and for reading and writing to the serial port). A sample screen of the client is depicted in Figure 6.

To operate the client, the COM port has to be set (in the lower right corner) and the port opened ("Connect"). There are fields for displaying raw bytes received, comments, and decoded data. In addition, there are fields for showing readings from eight slaves, together with a standard deviation of all values received.

There are preprogrammed fields to send almost all commands to the 1611, or the user can assemble his own command with up to three parameters (it is important in this case to set the *#pr* field). Due to sloppy programming (and bad Visual-C++ conventions) the display fields cannot be scrolled; however, new information always appears in the top. Fields can be reset by “Reset”.



|  | A/D-0 | A/D-1 | A/D-2 | A/D-3 | A/D-4 | A/D-5 | A/D-6 | A/D-7 |
|---|---|---|---|---|---|---|---|---|
| Read: | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 |
| Stdev: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|  | A/D-8 | A/D-9 | A/D-A | A/D-B | A/D-C | A/D-D | A/D-E | A/D-F |
|---|---|---|---|---|---|---|---|---|
| Read: | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 |
| Stdev: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Com Port: 9

**Figure 6.** A simple communications client.

## References

[1] AME, Inc., “AME8815 1.5A CMOS LDO,” http://www.ame.com.tw/English/Datasheet/ame8815.pdf

[2] Fairchild Semiconductor, “Fairchild P/N FAN2504,” http://www.fairchildsemi.com/pf/FA/FAN2504.html

[3] Future Technologies Devices International Ltd., “FTDI Drivers,” http://www.ftdichip.com/FTDrivers.htm

[4] Maxim, Inc., “MAX629 28V, Low-power High-Voltage, Boost or Inverting DC-DC converter,” http://www.maxim-ic.com/quick_view2.cfm/qv_pk/1705

[5] MaxStream, Inc., “XBee ZigBee OEM RF Module,” http://www.maxstream.net/products/xbee/xbee-oem-rf-module-zigbee.php

[6] Phillips Semiconductor, “The I$^2$C Bus Specification, Version 2.1,” January 2000, http://www.nxp.com/acrobat_download/literature/9398/39340011_21.pdf

[7] Sourceforge, “mspgcc – GCC toolchain for MSP430,” http://mspgcc.sourceforge.net/

[8] OLIMEX, Ltd., "JTAG-Tiny," http://www.olimex.com/dev/msp-jtag-tiny.html

[9] OLIMEX, Ltd., "MSP430H1611 Header Board," http://www.olimex.com/dev/msp-h1611.html

[10] Texas Instruments, "16-bit Ultra-low-power MCU MSP430F1611," http://focus.ti.com/docs/prod/folders/print/msp430f1611.html

[11] Texas Instruments, "16-bit Ultra-low-power MCU MSP430F2013," http://focus.ti.com/docs/prod/folders/print/msp430f2013.html

[12] Texas Instruments, "IAR Embedded Workbench Kickstart," http://focus.ti.com/docs/toolsw/folders/print/iar-kickstart.html

[13] Texas Instruments, "MSP430 USB Debugging Interface," http://focus.ti.com/docs/toolsw/folders/print/msp-fet430uif.html

[14] Texas Instruments, "MSP430F20xx Device Erratasheet," http://focus.ti.com/lit/er/slaz026h/slaz026h.pdf

[15] Texas Instruments, "MSP430x2xx Family User's Guide," http://focus.ti.com/lit/ug/slau144c/slau144c.pdf

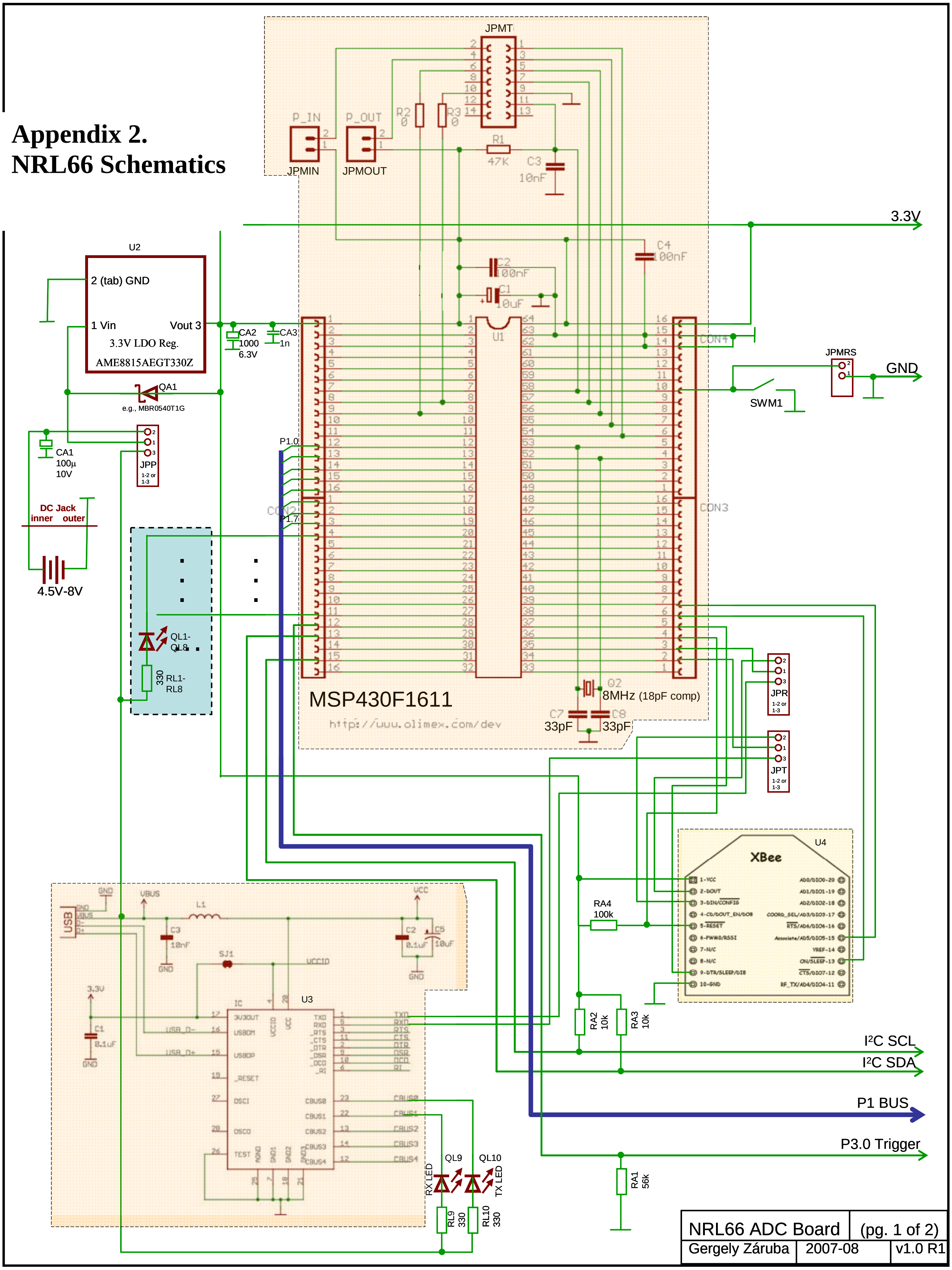
[16] Záruba, G.V, "Useful Stuff from Gergely Záruba," http://crystal.uta.edu/~zaruba/usefulsutff.html

## Appendix 1.   (7,4) Hamming Coding and  Decoding

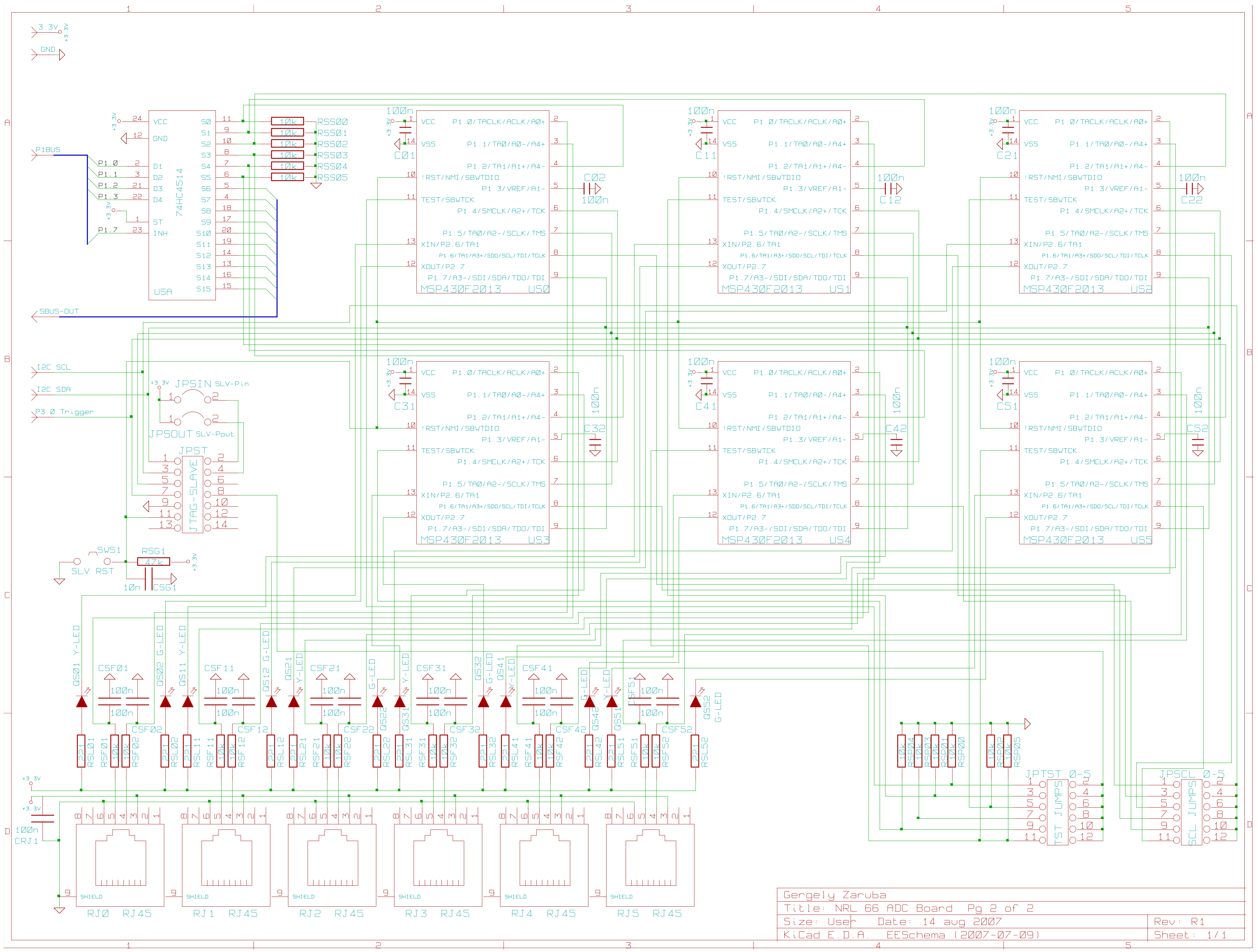
```
//Array of codes:
unsigned char HammingCode[16] =
    {
        0x00 , 0x0F , 0x13 , 0x1C ,
        0x25 , 0x2A , 0x36 , 0x39 ,
        0x46 , 0x49 , 0x55 , 0x5A ,
        0x63 , 0x6C , 0x70 , 0x7F };

//coding is simple:
void Convert2Code(unsigned char input)
{

    unsigned char p_low = HammingCode[input & 0x0F];
    unsigned char p_high = HammingCode[(input>>4) & 0x0F];
    SendByte(p_low);
    SendByte(p_high);
}


//decoding is a little more involved
// the first three functions are helper functions , the last one does the
decoding

char CorrectAndDecode(unsigned char code)
{
    unsigned char mask = 1;
    unsigned char n;
    char i;
    for (i = 0; i < 8; i++)
    {
        n = A_CodeToData (code ^ mask);
        if (n != -1)
            // Corrected it!
            return n;
        mask <<= 1;
    }
    // did not work
    return -1;
}

char A_CodeToData (unsigned char code)
{
    char i;
    for (i = 0; i < 16; i++)
    {
        if (code == HammingCode[i])
        {
            return i;
        }
    }
    // Not a code!
    return -1;
}

char CodeToData(unsigned char code)
{
    char n;
    n = A_CodeToData(code);
    if(n != -1)
        return n;
    n = CorrectAndDecode(code);
    return n;
}
```

```
int ByteToData(unsigned char low_byte, unsigned char high_byte)
{
    int highval=CodeToData(high_byte);
    int retval=CodeToData(low_byte);
    if((retval==-1)||(highval==-1))
        return -1;
    return(retval+(highval<<4));
}
```

# Appendix 2.
# NRL66 Schematics

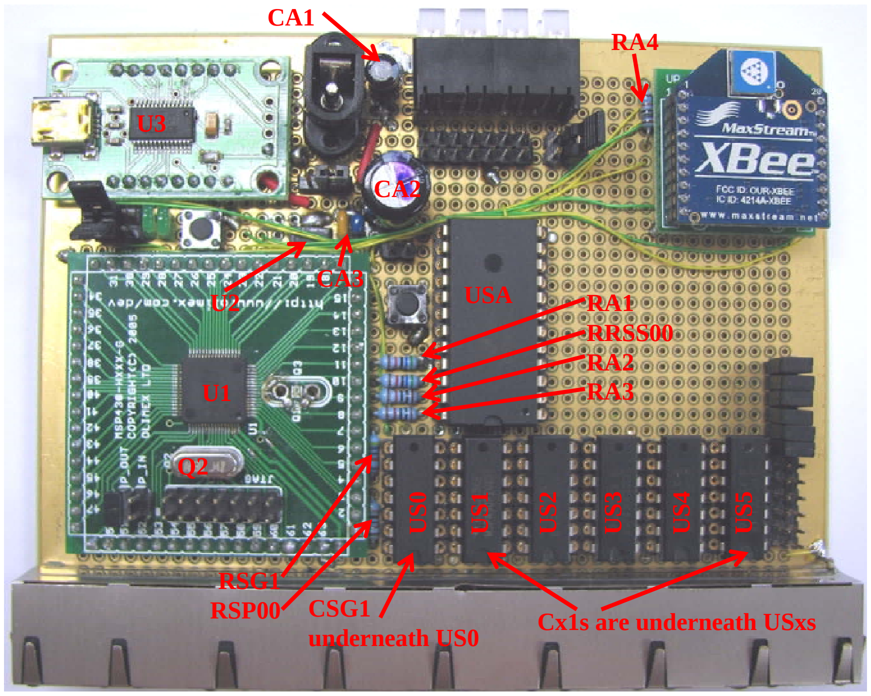


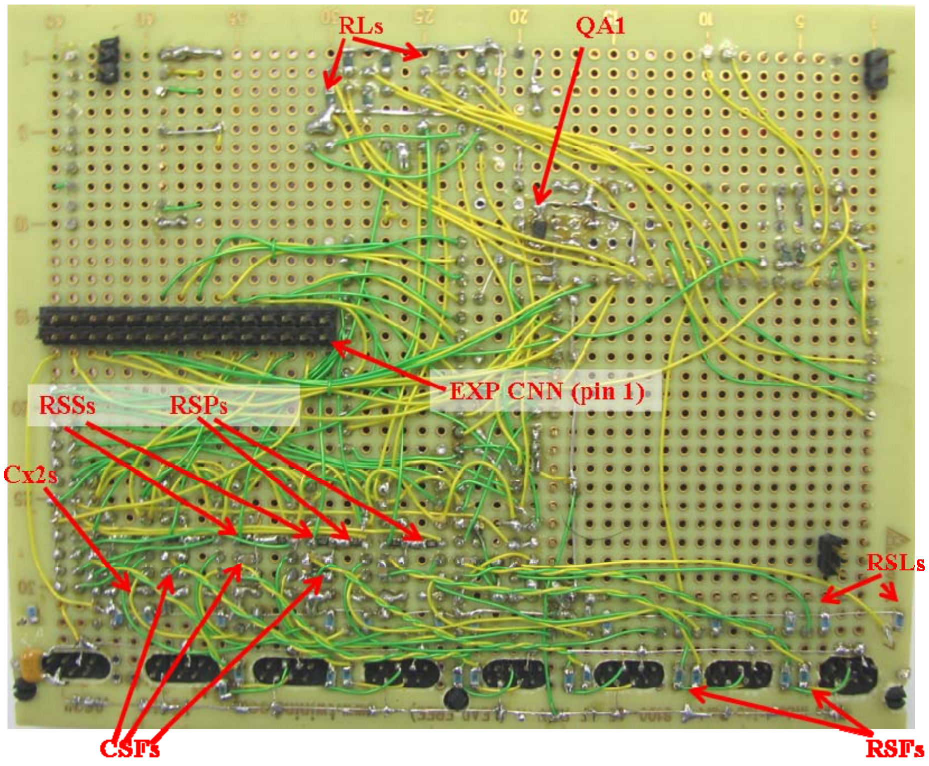
| NRL66 ADC Board | (pg. 1 of 2) | |
|---|---|---|
| Gergely Záruba | 2007-08 | v1.0 R1 |



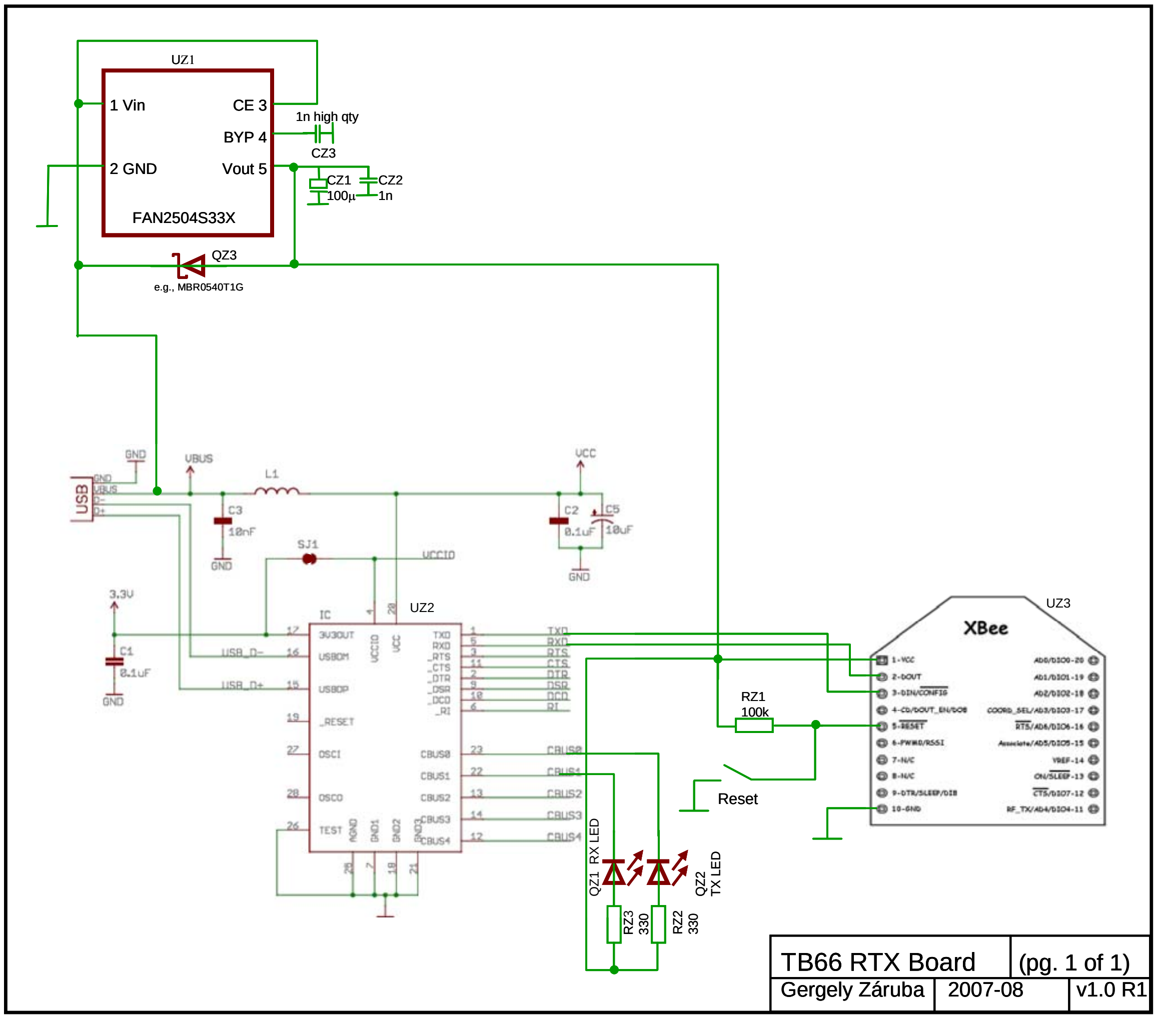
Gergely Zaruba
Title: NRL 66 ADC Board Pg 2 of 2
Size: User Date: 14 aug 2007 Rev: R1
KiCad E.D.A. EESchema (2007-07-09) Sheet: 1/1

## Appendix 3. ADC66 Component Locations





## Appendix 4.  TB66 Schematics



# Appendix 5. An ADC66 Daughterboard (DB66)

To extend the ADC66 with eight more ADC slaves, we have developed a piggy-back daughterboard (DB66) that connects to the ADC66 using the expansion connector. Figure 7 depicts the ADC66 combined with this daughterboard. In this appendix we are going to provide descriptions, pictures, and slave programming directions to the daughterboard. At the time of this writing, the RJ-45 connectors of the DB66 remain unconnected as signals to be processed need to be selected.

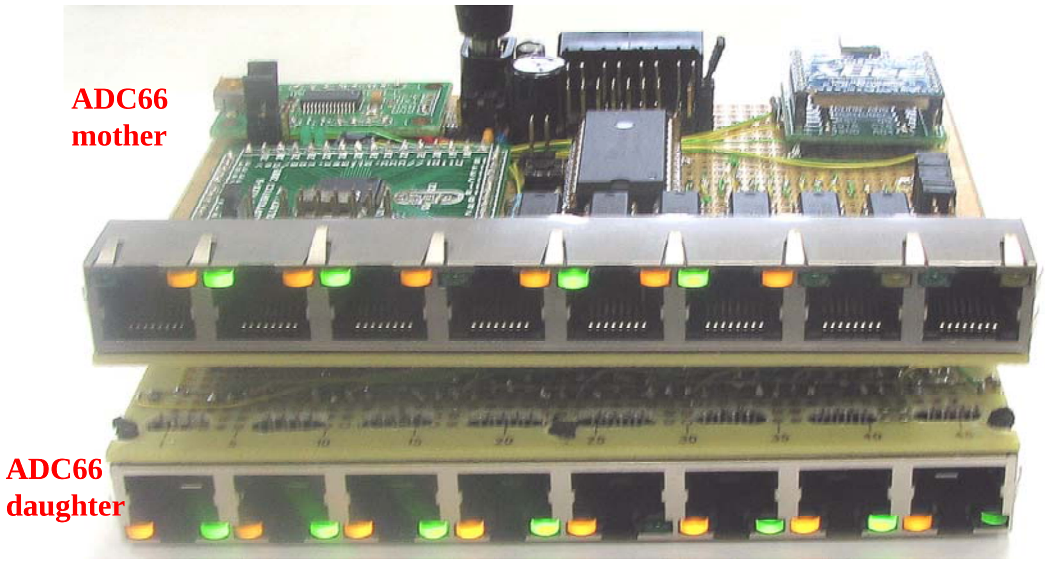


**Figure 7.** ADC66 with piggy-backed daughterboard.

## *Programming Slaves on the DB66*

Theoretically, slaves should be programmable through the ADC66 board by setting JPTST, JPSCL, JPDSCL, and JPDTST appropriately; however, it is recommended that when programming either set of slaves (the ones on the ADC66 or the ones on the DB66) the DB6 to be disconnected from the ADC66. On the DB66, the SCL and TST jumper blocks have been replaced by dip switches. During regular operation all JPDSCL switched need to bi in their "on" posisiont and all JPDTST switches need to be in their "off" position (as depicted in Figure 8). When programming slaves only one switch in JPDSCL and only one (the correspondingly numbered) switch in JPDTST can be "on". JPDSCL1/JPDTST1 will program US6, JPDSCL2/JPDTST2 will program US7, …, and JPDSCL8/JPDTST8 will program US13. As the DB66 is disconnected from the ADC66, JPDIN needs to be jumpered (and JPDOUT needs to be off). JPDB is the JTAG connector (which has the same functionality and connections as JPST).

## *Connecting Analog Input signals*

At the time of this writing it was unclear what signals the new slaves are going to be processing. Thus their respective inputs have been connected to the ground using a 100nF capacitor and connected (using twisted pairs) to the unused upper part of the DB66 (as depicted by IN6-, IN6+, …, IN13-, IN13+ in Figure 9). Additional signal conditioning active or passive components may be placed here and their inputs connected to RJ6,…,RJ13.

## Remarks

The heat sinks on the corners of the ADB66 are only used as distance holders so the board lays flat when turned upside-down (in its install position). Only the LEDs of RJ6 to RJ13 are connected, the rest of the pins can be connected as desired. The expansion connector on the bottom side was manufactured from a floppy-drive cable and when soldered, it switched sides of pins (see pin1 designation of connector and solder anchor in Figure 9). The schematic to the DB66 is similar to the schematic of the ADC66 and can be seen in Figure 10.

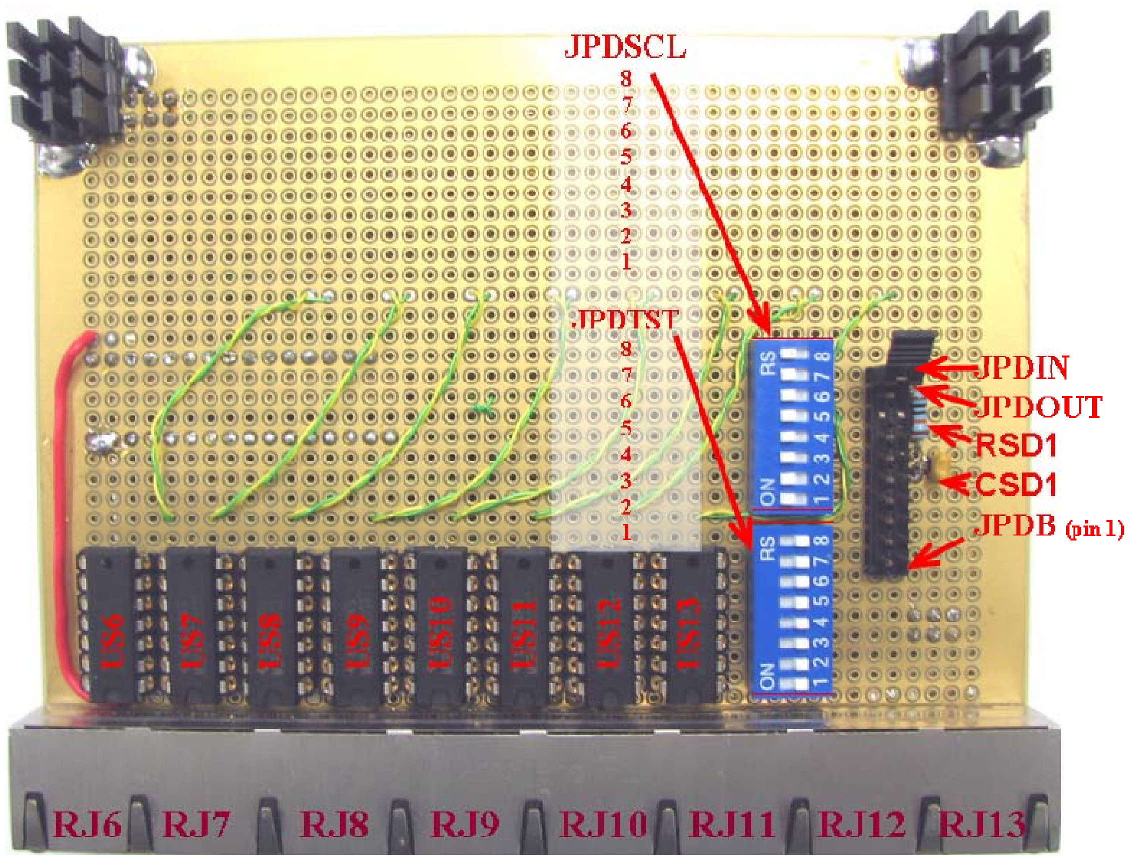


**Figure 8.** Top side of DB66.

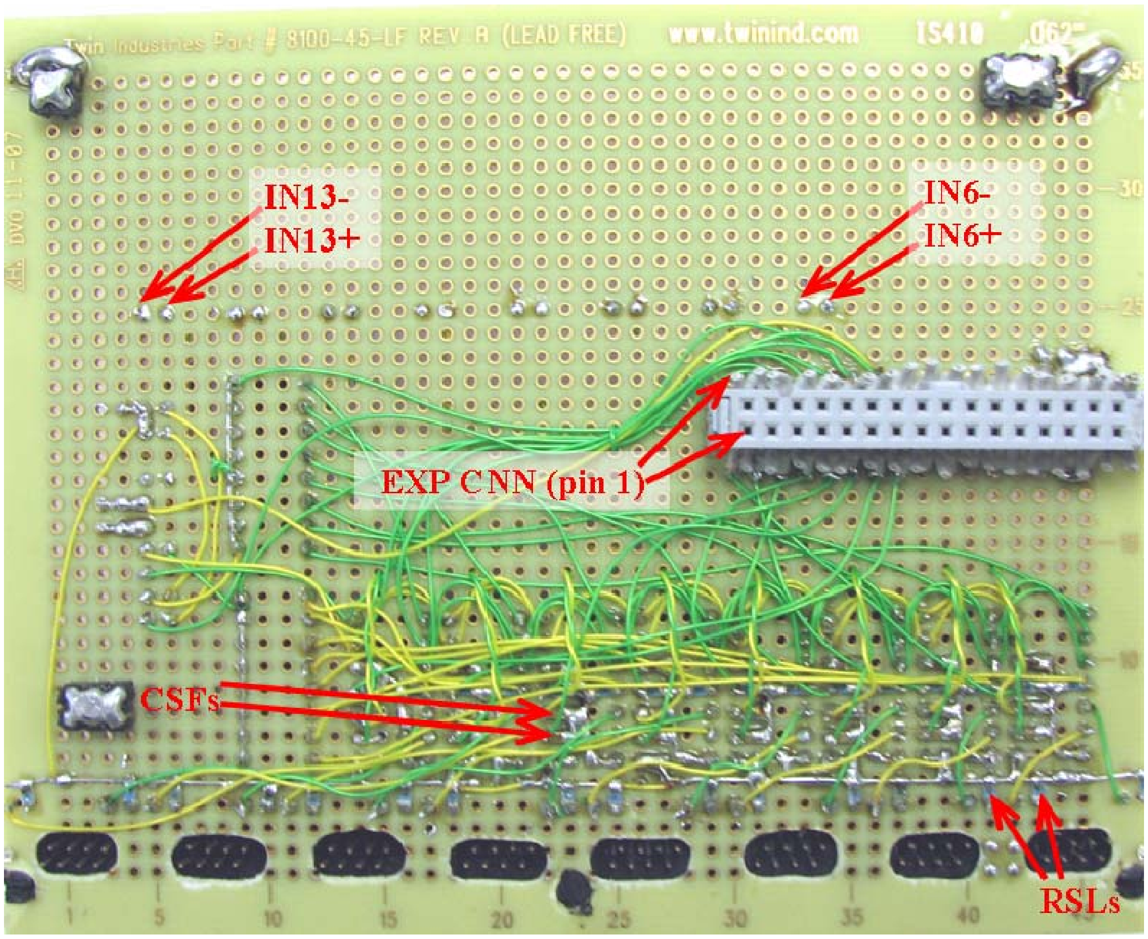


**Figure 9.** Bottom side of DB66.

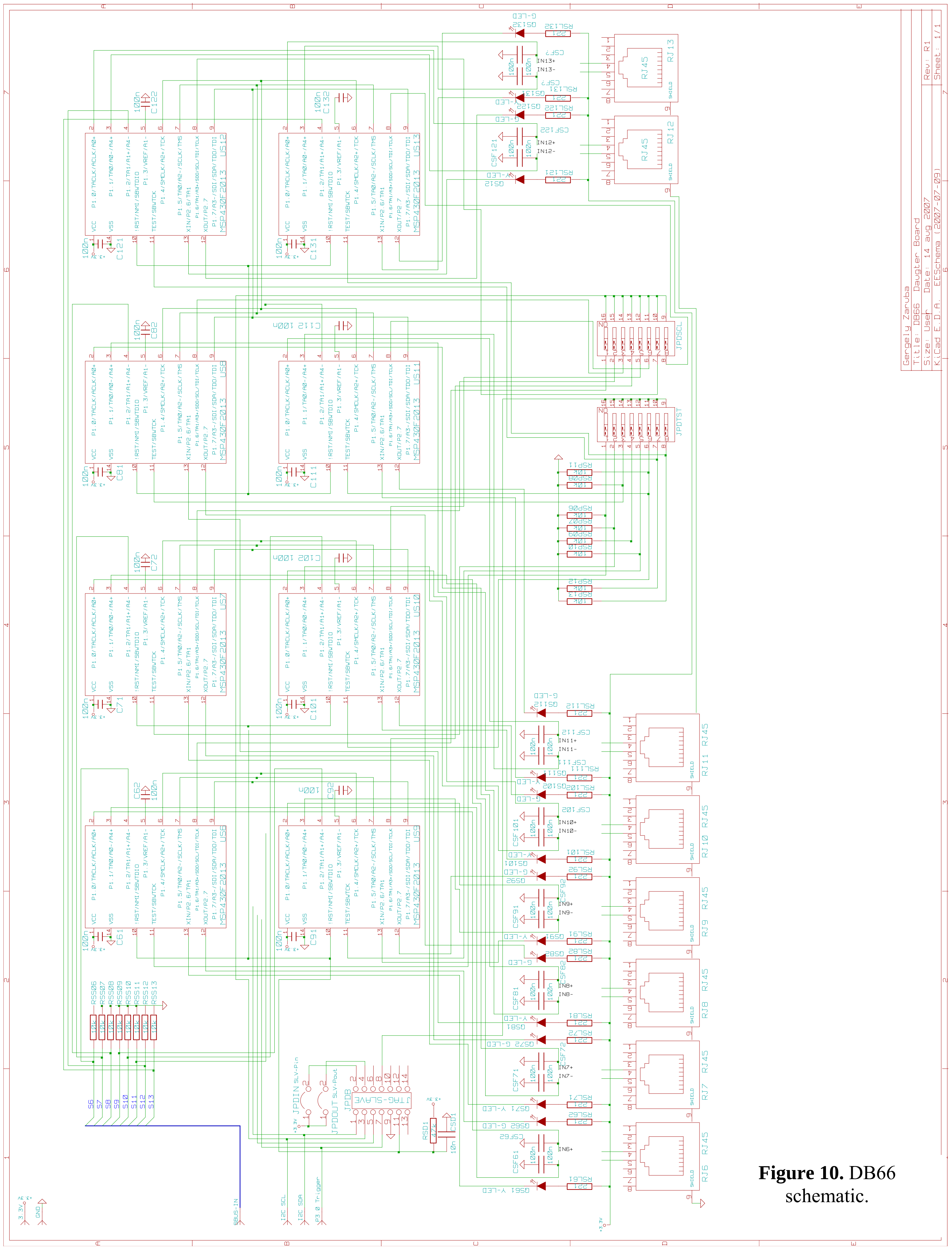


**Figure 10.** DB66 schematic.