SG2650 board

Ivan Z. aka 'Giles'
gilesgoat@gilesgoat.com

March 22, 2014

Contents

Abstract

Long time ago a friend of a friend of mine gave me a bag full of really old chips, some of them were some curious interesting CPUs. Years later an idea to preseve the memory and knowledge of those unusal CPUs came in mind and so I did start a mission to create small but still useful boards with those CPUs. This is one of those boards, a simple one based around the Signetics 2650.

1 Introduction

The SG2650 board is a single board computer based around the SG2650 CPU.

The board is a standard 160x100 Eurocard board with an DIN41926 64 pin connectors designed to fit in a standard 19 inches 3U rack.

2 Constructional details of board

The board is a standalone computer with a CPU, RAM, ROM,an UART for RS232 communication, and IDE port and a DIN41926 connector for interfacing with other components.

The UART provides 1 serial channel which is then connected to a levels translator to provide correct RS232 level, two special CPU pins named SENSE and FLAG are also connected to the levels translator and provide an additional software driven serial communication port.

One DB9 male connector is present on the main bezel the two serial port as well as a reset pushbutton switch.

A standard 40 pins IDE (PATA) connector is also present on the board for connection with some sort of hard drive unit.

The main logic functions are provided via a single CPLD chip to reduce components count.

The board also contains two CAN clock oscillators, one to provide the main 1 Mhz system clock on the board and one to provide an 1.8432 Mhz used for baud rate generation.

2.1 The power supply

The board requires a single power supply at 5V (VCC) board consumption been not measured but it's supposed to be around various hundreds of miliampers. The required tensions for the RS232 interface are derived internally from the 5V via charge pump methods by the MAX232 chip.

2.2 The CPU

The CPU is a Signetics 2650 clocked at 1 Mhz.

As the board is very minimal and the component load is small no buffering is present, if other cards have to be connected to it the added card must contain its own buffers for the required lines.

2.2.1 The CPU clock

The CPU clock is generated by the CPLD via a divisor, the main oscillator clock is divided as necessary to generate 1 Mhz.

The frequency of 1 Mhz been chosen because this is also the maximum frequency the UART can cope with.

The clock is a standard TTL level clock with a duty cycle of 50 percent, this duty cycle is guaranteed by a flip flop acting as a divide by two clock shaper.

NOTE : in this particular implementation a 16 Mhz CAN oscillator is first divided by 8 and then further by 2 by the clock shaper to get a perfect 50 percent duty cycle 1 Mhz clock.

2.2.2 The CPU reset

The CPU reset is generated by the CPLD as well and is triggered by power up or the panel reset button.

At power up until capacitor C10 is charged via R3 a low level is forced on the MReset pin, the CPU reset pin is hold low as long as this condition persists.

When MReset returns high the CPU reset pin is released and set to logic '1' after four CPU clock cycles.

When the pushbutton P1 is pressed this fully discharges the capacitor C10 restarting the whole thing above once released.
[1]

2.2.3 The CPU addressing space

The CPU has a maximum addressing space of 32K which is split between ROM addressing space and RAM addressing space.

At power up or immediately after a reset the first 4K of addressing space are designed as ROM and the remaining of the addressing space as RAM.

Table 1: CPU address space after power up or reset

Address range	Memory
0000H - 0FFFH	ROM
1000H - 7FFFH	RAM

The RAM is implemented as a single chip 43256 32k x 8 bits, while the ROM can be selected via jumper J2 to be a 6264 (8K x 8) or a 2732 (4K x 8) chip.

The first 4K of the addressing space can be dedicated to RAM by writing into the CPLD "map register".

The ROM is designed to contain the bootstrap program.

2.2.4 The CPU I/O

The I/O employed used Extended mode only of the CPU which means an I/O address too has to be supplied.

[1] As the RAM is STATIC and the bootloader does NOT touch the ram contents except its own data area a reset via pushbutton should not alter RAM contents in any way, this is very useful for debugging.

The I/O address space is only partially decoded, lines ADR4 and ADR3 are used to identify four different I/O areas while lines ADR0 - ADR2 are used by the IDE interface to address the IDE registers during an IDE cycle.

The space is partitioned as by the following table :

Table 2: I/O space partitioning

ADR4 - ADR3	Meaning
0 0	CPLD memory map register
0 1	UART
1 0	IDE Low byte (D0 .. D7)
1 1	IDE High byte (D8 ... D15)

2.2.5 The CPU SENSE and FLAG lines

The SENSE and FLAG lines are connected to the RS232 level translators, this allows them to be used as a software programmed serial communication port. The lines after the RS232 translators are routed to connector CON4

Table 3: CON4 pinout

Pin	Signal	Direction
1	Flag	Output (RS232 level)
2	Sense	Input (RS232 level)
3	GND	Ground

2.3 The UART

The UART is a 6850 ACIA clocked at 1 Mhz. Its transmit and receive clock are tied together and are supplied a 16 x Baudrate clock from the CPLD. Only the line TX and RX are connected to the RS232 level translators, CTS and RTS lines are not used.

The RS (register select) pin is connected to ADR0.

The UART is selected via Extended I/O operations any time ADR4-ADR3 are as “01”.

2.3.1 UART registers

The UART has 4 registers mapped as follow

Table 4: UART registers mapping

ADR2	I/O Operation	Meaning
0	Write	Control register
0	Read	Status register
1	Write	Transmit register
1	Read	Receive register

For convenience here a quick look at the Control and Status register.

Table 5: UART Control register

Bit	Meaning
0	Counter Divide Select 1, (CR0)
1	Counter Divide Select 2, (CR1)
2	Word Select 1, (CR2)
3	Word Select 2, (CR3)
4	Word Select 3, (CR4)
5	Transmit Control 1, (CR5)
6	Transmit Control 2, (CR6)
7	Receive Interrupt Enable (CR7)

Table 6: UART Statusl register

Bit	Meaning
0	Receive Data Register Full, (RDRF)
1	Transmit Data Register Empty, (TDRE)
2	Data Carrier Detect, (DCD)
3	Clear To Send, (CTS)
4	Framing Error, (FE)
5	Receiver Overrun, (OVRN)
6	Parity Error, (PE)
7	Interrupt Request (IRQ)

2.3.2 The baud rate clock

The UART requires a baudrate clock which must be 16 times the desired baud rate, this clock is generated by a divisor internal to the CPLD starting from a 1.8432 Mhz clock.

The 1.8432 Mhz clock is generated by a CAN oscillator.

The supplied UART clock is set to 16 times 9600 so is at 153600 Hz via an internal 12 divisor counter present inside the CPLD.

2.3.3 The uart interfacing

A closer look at the 6850 timings shows that it's possible to connect it directly to the SG2650 by simply supplying it with a inverted (NOT) clock, in such a way it turns out its bus timings are going to coincide with the SG timings provided that CPU clock and UART clock are the same frequency.

2.3.4 The uart TX and RX lines

After the TTL to RS232 translator the transmit and receive lines are routed to connector CON2 with the following pinout :

Table 7: CON2 pinout

Pin	Signal	Direction
1	Flag	Output (RS232 level)
2	Sense	Input (RS232 level)
3	GND	Ground

2.4 The IDE interface

An IDE interface is present on board, this allows IDE (PATA) devices to be connected to it.

The IDE interface is designed to support PIO mode only I/O (no support for DMA) and is fundamentally a reworked P.R.I.D.E interface.

The CPLD posses two registers called "IDE Low Byte" and "IDE High Byte", the access modalities are the usual ones.

In the case of a WRITE operation first the IDE High Byte register has to be written followed by a write on the IDE Low Byte register with ADR0 .. ADR2 containing the value of the IDE register you wish to use.

Table 8: IDE Write Cycle

Action	Effect
1. Extend I/O write on HI register	Data HI is latched
2. Extend I/O write on LOW register	16 bits data HI+LOW ready ADR0..2 selects IDE register IDE write cycle starts

In the case of a READ operation first the IDE Low Byte register has to be read with ADR0 .. ADR2 containing the value of the IDE register you intend to read followed by a read of the IDE High Byte register.

Table 9: IDE Read Cycle

Action	Effect
1. Extend I/O read on LOW register	ADR0..2 selects IDE register IDE read cycle starts Data HI is latched Data LOW is read
2. Extend I/O read on HI register	Data HI is read

ONLY when the IDE Low Byte register is being used a IDE_WR or IDE_RD signal together with an IDE_CS0 is generated, when IDE High Byte register is being used all those signals are kept inactive.

In this implementation IDE_CS1 is hardwired to logic level '1' .

2.4.1 The IDE connector

A standard 40 pin IDE connector header is present on the board, this connector follows the standard IDE pinout as shown below :

Table 10: IDE 40 pins connector assignement

Pin	Name	Pin	Name
1	Reset	2	GND
3	D7	4	D8
5	D6	6	D9
7	D5	8	D10
9	D4	10	D11
11	D3	12	D12
13	D2	14	D13
15	D1	16	D14
17	D0	18	D15
19	GND	20	key
21	DMARQ	22	GND
23	/DIOW	24	GND
25	/DIOR	26	GND
27	IORDY	28	CSEL
29	/DMACK	30	GND
31	INTRQ	32	/IOCS16
33	DA1	34	PDIAG
35	DA0	36	DA2
37	/IDE_CS0	38	/IDE_CS1
39	/ACTIVE	40	GND

A standard 40 pins IDC cable is supposed to be used to connect it to a hard drive or other storage unit.

2.5 The CPLD map register

This is simply a write only 1 bit register that can be accessed by an Extended I/O operation when ADR4-ADR3 are as “00” .

As there is no data bit connected to it instead an address line ADR2 is used for that so the value of ADR2 is used to assign the value to this register bit.

When the register bit is set to ‘1’ then the ROM totally disappears (deselected) from the address space, making the full 32K of available address space become RAM.

When the register bit is set to ‘0’ the first 8K of addressing space become ROM and the remaining 24K are RAM.

At power up or after a reset the register bit is set to ‘0’ .

Table 11: Map register write access

ADR4-ADR3	ADR2	Meaning
“00”	“0”	0000H - 0FFFH is ROM
“00”	“1”	0000H - 0FFFH is RAM

We remind that 1000H - 7FFFH are always RAM.

2.6 The main edge connector

The main edge connector is the standard DIN 41612, see the table for the pins assignements.

Table 12: Main edge connector

Pin	Row A	Row C
1	adr6	adr5
2	adr4	adr7
3	n.c.	adr11
4	n.c	adr10
5	n.c.	adr9
6	VCC	VCC
7	adr14	adr13
8	n.c.	CpuCLK
9	n.c.	adr12
10	n.c.	adr3
11	n.c.	adr8
12	n.c.	n.c.
13	n.c.	d0
14	n.c.	d1
15	n.c.	d2
16	n.c.	d3
17	n.c.	d4
18	n.c.	d5.
19	n.c.	d6
20	n.c.	d7
21	n.c.	adr0
22	n.c.	n.c.
23	n.c.	opreq
24	n.c.	negR-W.
25	n.c.	WRP
26	n.c.	MnegIO
27	n.c.	n.c.
28	n.c	adr1
29	n.c.	MRESET
30	n.c.	BaudCLK
31	adr2	baudclk16
32	GND	GND

Table 13: Signals meaning

Signal name	Type	Meaning
adr14 . . . adr0	Output	Address lines
d7 . . . d0	Input/Output	Data lines
opreq	Output	Operation Request
negR-W	Output	Read (0) or Write (1)
WRP	Output	Write Pulse
MNegIO	Output	Memory (1) or IO (0)
CpuCLK	Output	CPU Clock 1Mhz
MRESET	Output	Master Reset (active 0)
BaudCLK	Output	Baudrate Clock
baudclk16	Output	16 X Baudrate Clock
VCC	Input	+5v Power Supply
GND	Input	Power Supply Ground (0 V)

“n.c.” means a not connected pin.
For more details about the signals please consult the SG2650 CPU User Manual.

2.7 JTAG connector

A 10 pins IDC type connector is present containing JTAG signals for programming of the CPLD. The connector is designed to fit the XILINX Parallel 3 Upload Cable.

Pin number 1 of the connector can be connected to the board VCC via jumper J1, normally this pin is NOT connected.

Table 14: JTAG connector pins assignements

Pin	Signal	Pin	Signal
1	VCC	2	GND
3	TCK	4	TDO
5	TDI	6	TMS
7	n.c	8	n.c.
9	n.c.	10	n.c.

"n.c." means a not connected pin.

2.8 The DB9 RS232 connector

On the front panel a male DB9 connector is present for the RS232 port, this is for connection to a computer or terminal to access the board monitor program. The pinout of the connector is as following :

Table 15: Front panel DB9 serial connector

Pin	function	direction
1	not used	n.a.
2	RXD	input
3	TXD	output
4	not used	n.a.
5	GND	ground
6	not used	n.a.
7	not used	n.a.
8	not used	n.a.
9	not used	n.a.

3 Jumpers settings

Two jumpers are present in the card, they are called J1 and J2.

Jumper J1 when in ON (inserted) position connects the VCC of the board to the VCC line (pin 1) of the JTAG connector, this allows to supply power to the upload cable from the board or to the board from the cable.

Jumper J2 when in ON (inserted) position supplies VCC to the pin 26 of the ROM socket, this is designed for a 2732 4Kx8 EPROM.

Table 16: Main Board jumpers J1 and J2

On : jumper inserted, Off : no jumper

Jumper	Position	Meaning
J1	On	Vcc to JTAG Pin 1
J1	Off	JTAG Pin 1 disconnected
J2	On	Vcc to ROM pin 26 (2732 used)
J2	Off	ROM pin 26 disconnected

Jumpers are normally both in OFF position (not inserted).
Check the pictures for jumper locations and position.

4 Schematics

We have here the fulll board schematics, there are various sheets such as :

- Project Root sheet
- CPU, RAM, ROM and CPLD
- UART and serial port
- IDE interface
- Main connector

The choice of components been around the idea to keep the number of chips low and what I already had around and simplicity of constructing such a board with home technology.

Of course different choices and even better optimisations could be done. The choice of the particular format and connector also been dictated by the will of making it fit inside a standard 19 inches rack therefore the 160 x 100 mm 3U Eurocard standard format been chosen.

All been manually constructed, PCB been designed, developed, etched and drilled manually as well as the construction and design of the front panel. At the time this board been built the Xilinx CPLD XC8536 was still available, for a new project you should replace it with the 3.3V version and add a voltage regulator for its power supply.

Figure 1: Project root

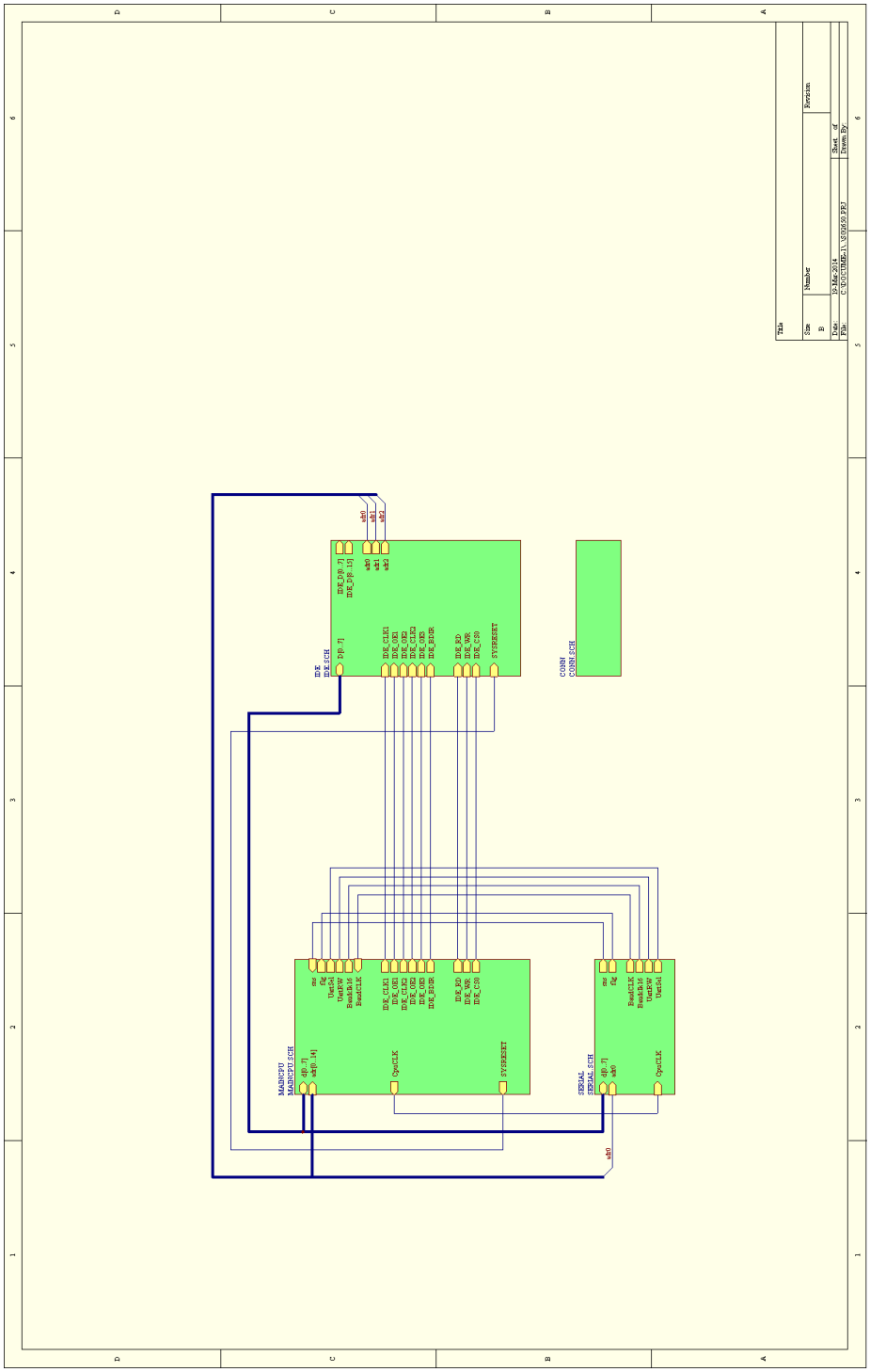


Figure 2: The CPU with RAM, ROM, CPLD and reset

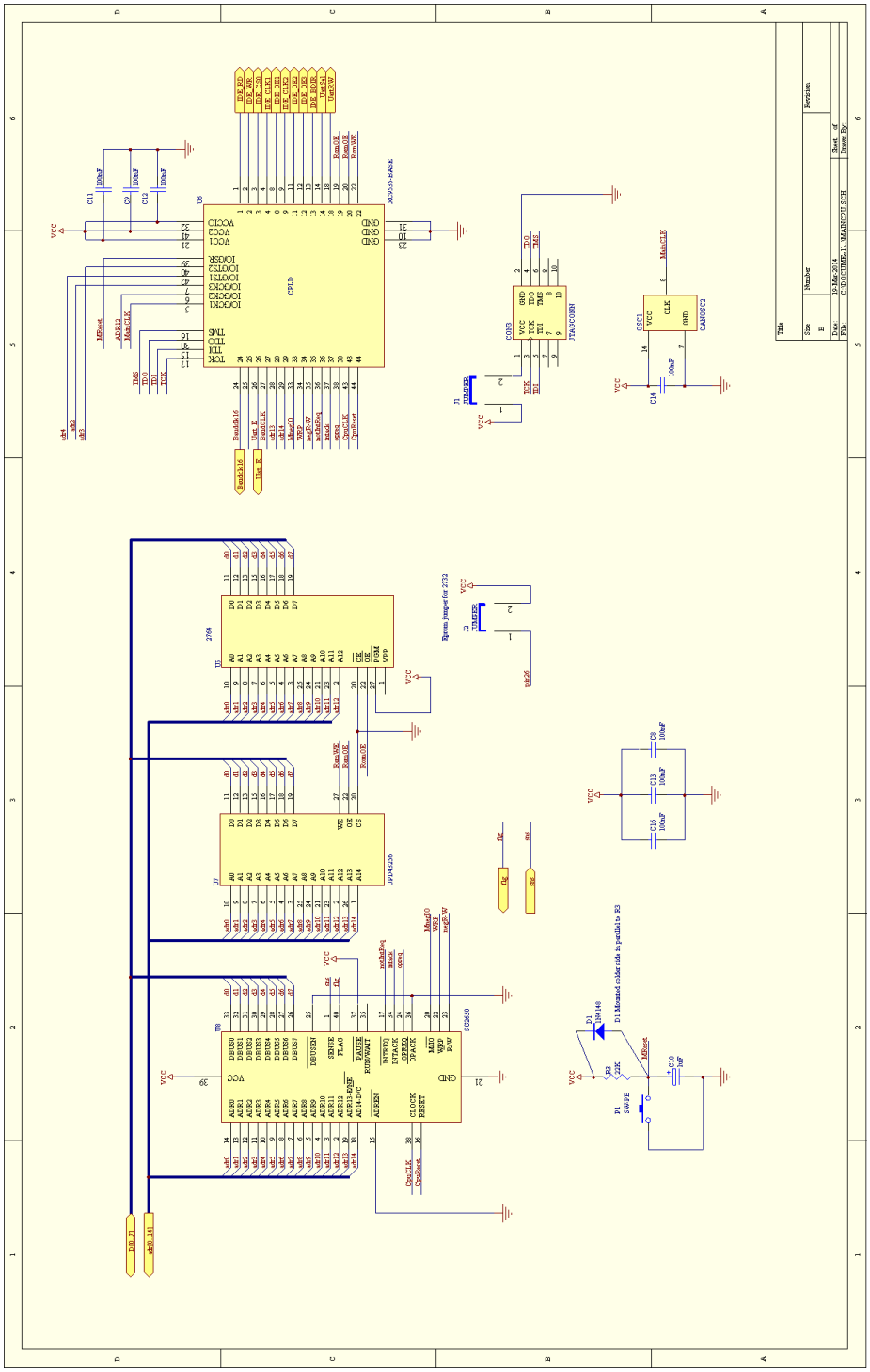


Figure 3: The UART and the RS232 port

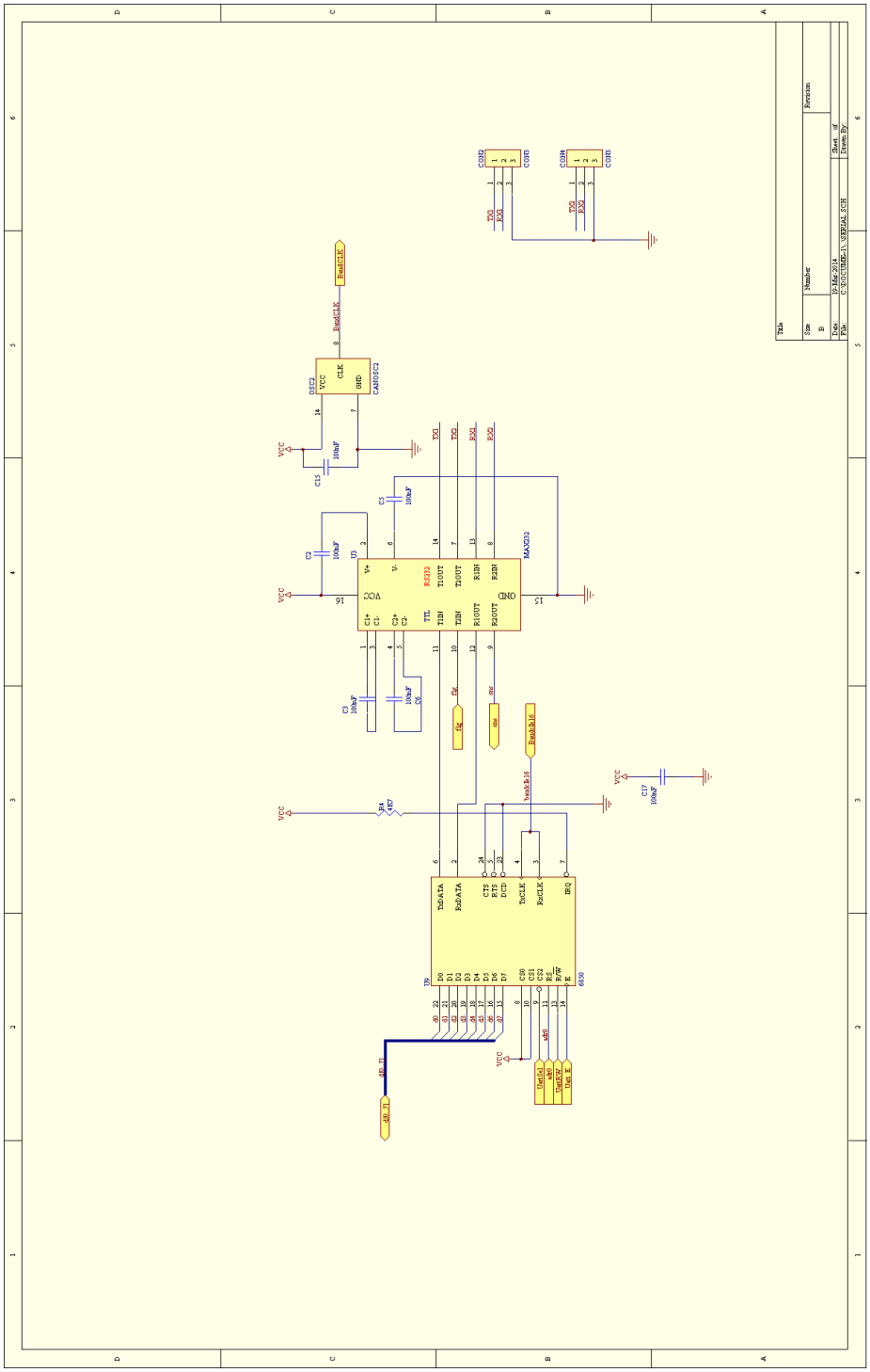


Figure 4: The IDE interface and connector

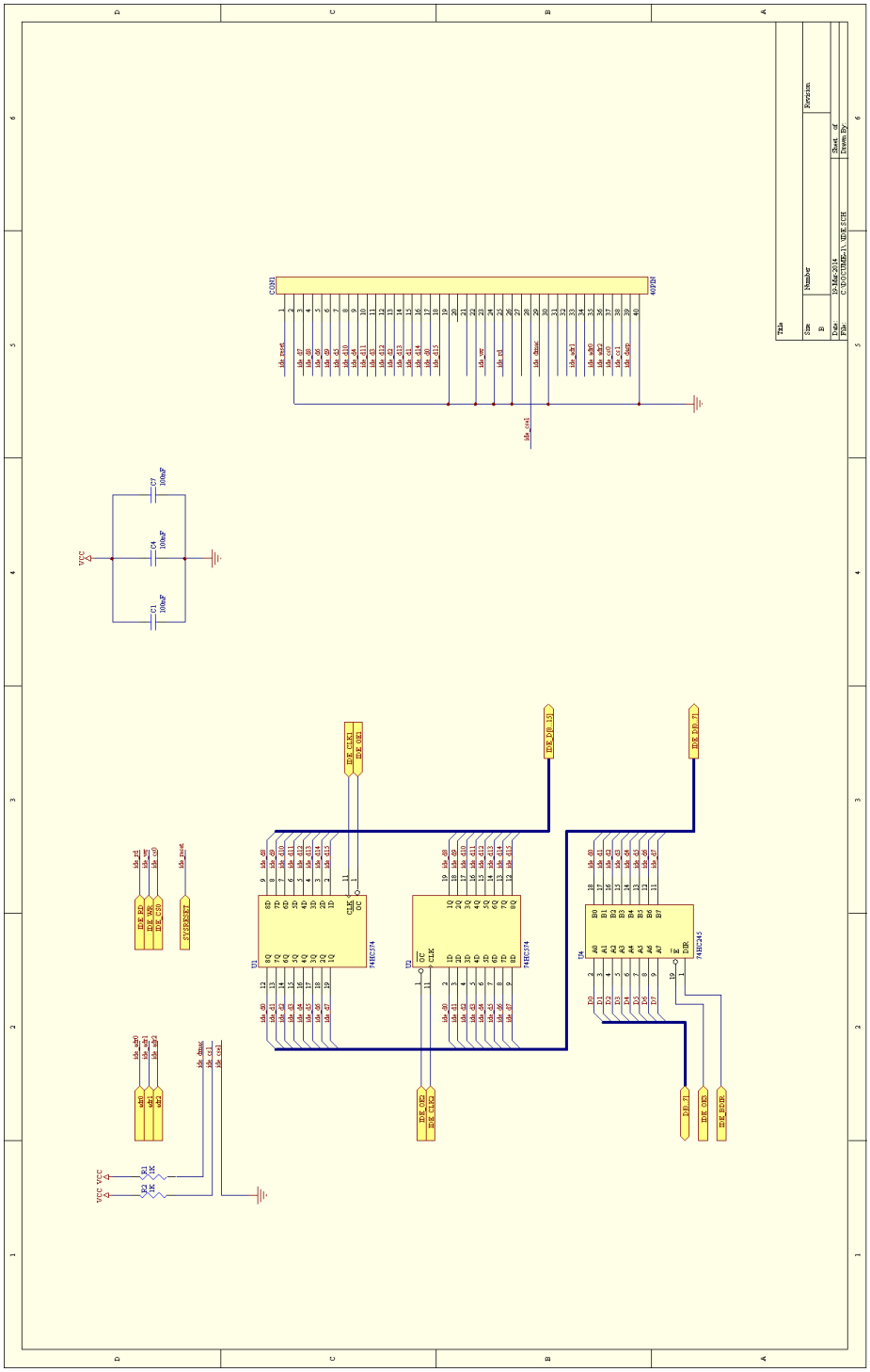
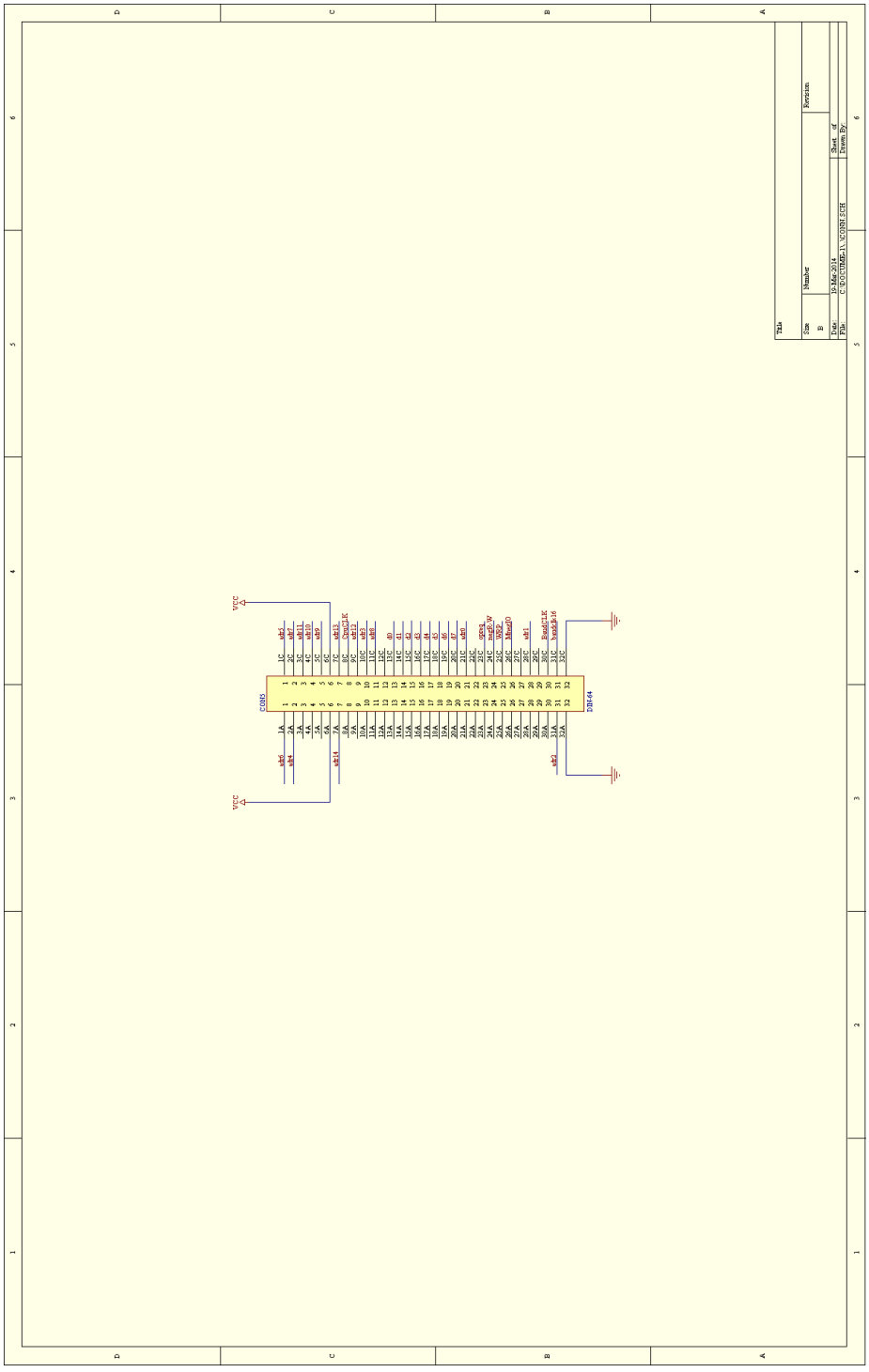


Figure 5: The main DIN 4192 connector



5 The PCB and components list

Please note those images are just for reference roundoff errors in the printing and conversion process make them look lightly wrong in places (tracks touching each other and some misalignments NOT present in the real print).

Warning the PCB shown contains a few little errors and some manual corrections via wire been done after, also the IDE interface never been tested yet.

5.1 The components list

Part	Used	PartType	Designators
1	2	1K	R1 R2
2	1	1N4148	D1
3	1	1uF	C10
4	1	4K7	R4
5	1	22K	R3
6	1	40PIN	CON1
7	1	74HC245	U4
8	2	74HC574	U1 U2
9	16	100nF	C1 C2 C3 C4 C5 C6 C7 C8 C9 C11 C12 C13 C14 C15 C16 C17
10	1	2764	U5
11	1	6850	U9
12	2	CANOSC2	OSC1 OSC2
13	2	CON3	CON2 CON4
14	1	DIN-64	CON5
15	1	JTAGCONN	CON3
16	2	JUMPER	J1 J2
17	1	MAX232	U3
18	1	SG2650	U8
19	1	SW-PB	P1
20	1	UPD43256	U7
21	1	XC9536	U6

Figure 6: Top Layer

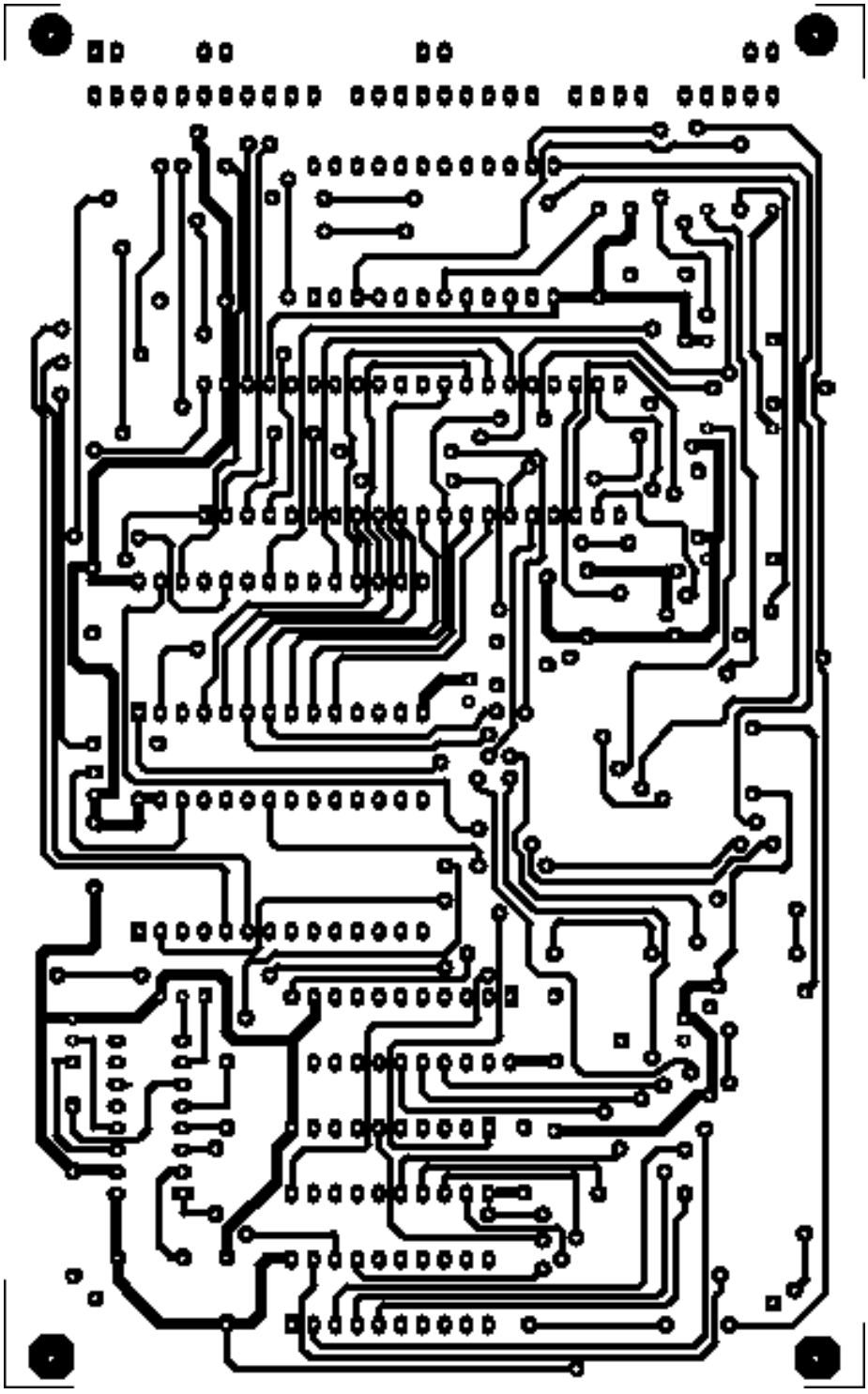


Figure 7: Bottom Layer

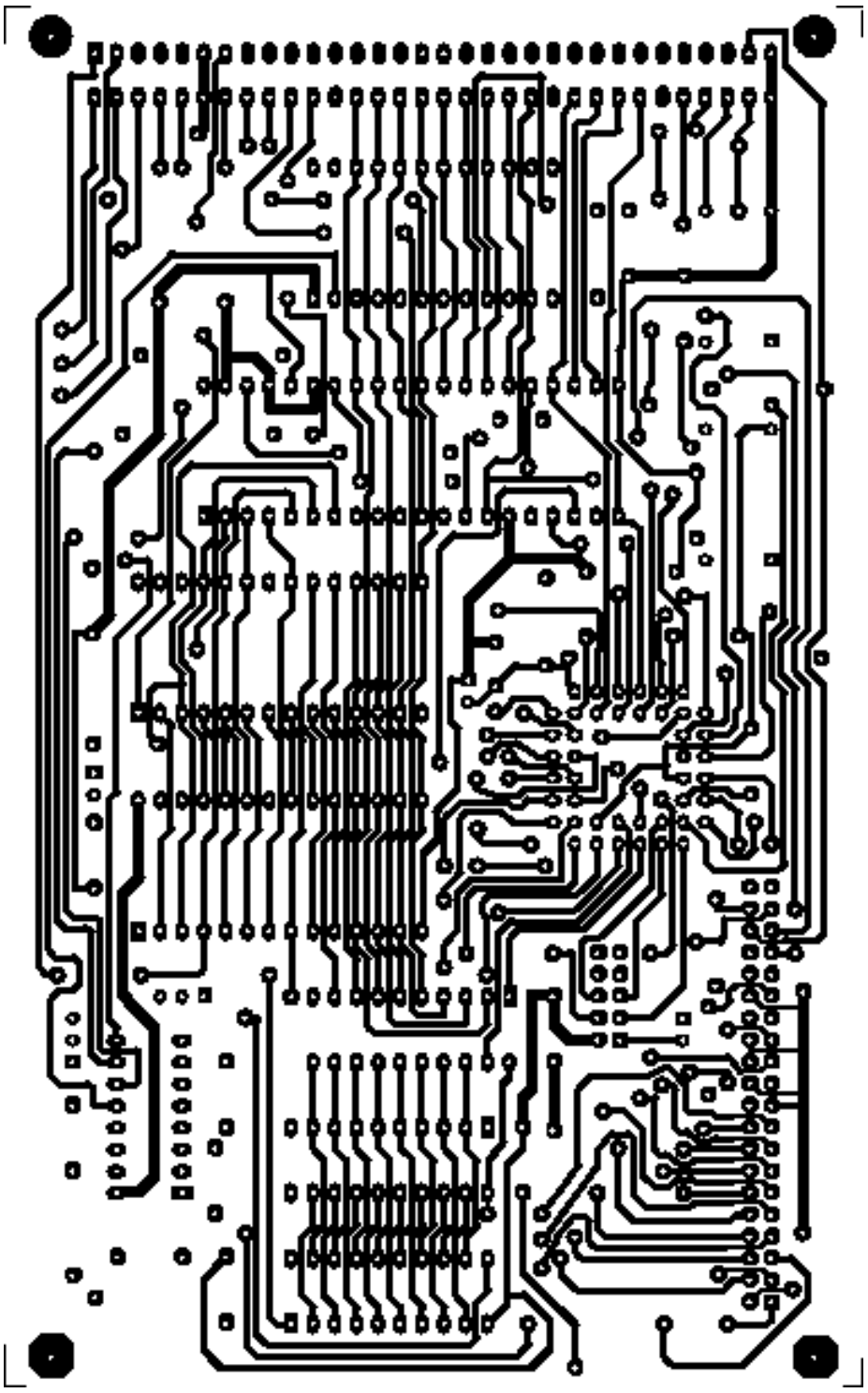
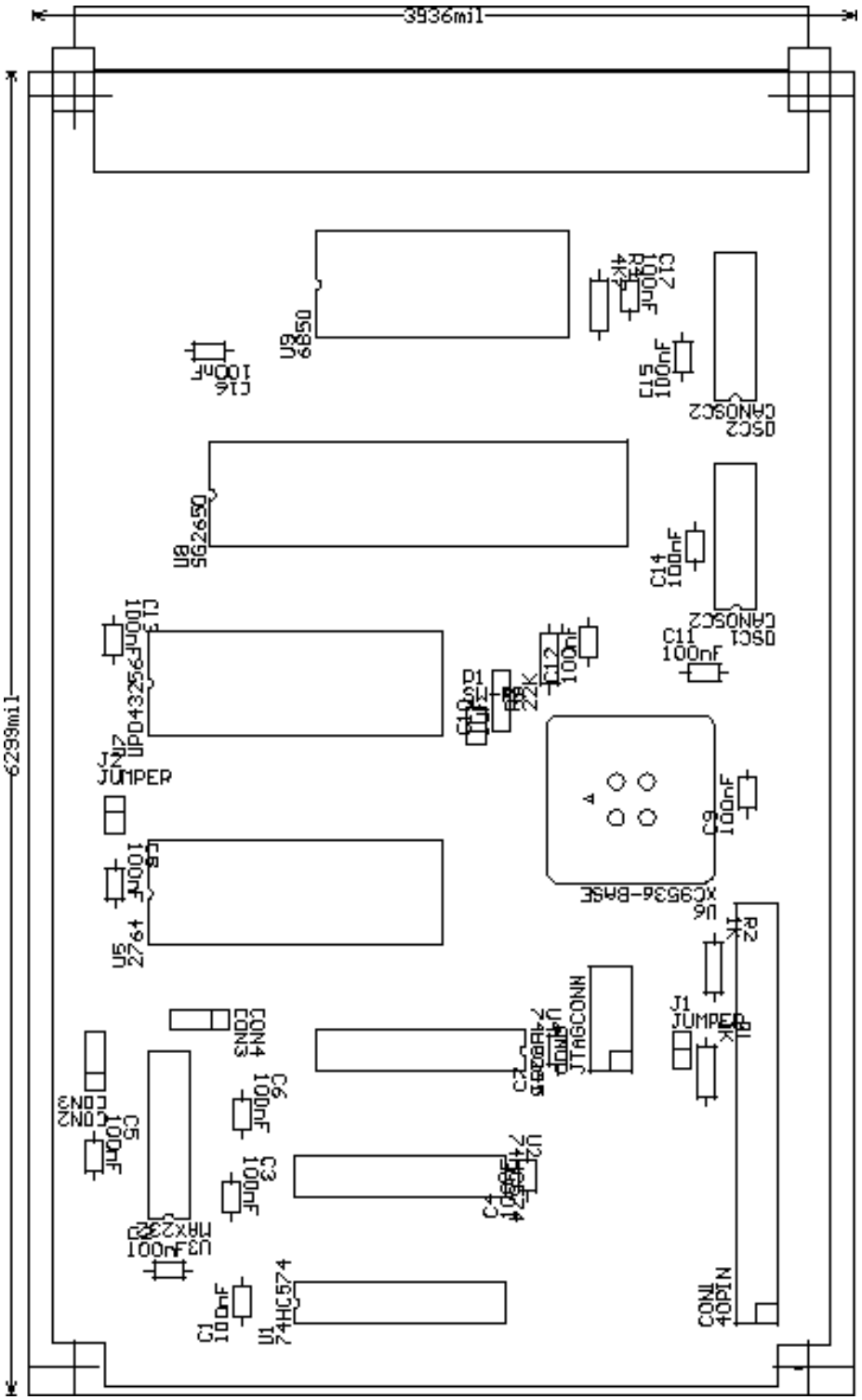


Figure 8: Silkscreen



6 CPLD VHDL source

Here follows the complete listing of the VHDL source that makes up the CPLD.

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:20:09 03/03/2012
-- Design Name:
-- Module Name:    SG2650_Logic - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_MISC.ALL;

entity ClockDivider is
    generic ( widthBits : integer := 2 );--default value is 2
    Port ( clkin : in STD_LOGIC;
               div : in STD_LOGIC_VECTOR (widthBits downto 0);
               clkout : out STD_LOGIC;
               negreset : in STD_LOGIC );
end ClockDivider;

architecture Behavioral of ClockDivider is

signal zeroCount : STD_LOGIC;
signal counter : STD_LOGIC_VECTOR (widthBits downto 0);
signal tmpclk : STD_LOGIC;
signal tmpZero : STD_LOGIC;

begin

     -- the first step is a divide by div counter
     DIVIDER: process ( clkin, negreset )
     begin
       if (negreset = '0') then
             counter <= (others => '0');
       elsif ( clkin'event and clkin='1') then
         if ( zeroCount = '0') then
             counter <= div;
         else
              counter <= counter-1;
         end if;
        end if;
     end process;

    zeroCount <= OR_REDUCE (counter);
    tmpZero <= zeroCount;
```

```
    -- the second step is a divide by 2 to get a 50% duty cycle
    SHAPER: process ( tmpZero, negreset )
    begin
     if (negreset = '0') then
       tmpclk <= '0';
     elsif ( tmpZero'event and tmpZero='1') then
       tmpclk <= not tmpclk;
     end if;
    end process;

    -- 50% duty cycle
    clkout <= tmpclk;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_MISC.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SG2650_Logic is
    Port (
              -- IDE Interface control signals
              IDE_RD : out  STD_LOGIC;
           IDE_WR : out  STD_LOGIC;
           IDE_CS0 : out  STD_LOGIC;
           IDE_CLK1 : out  STD_LOGIC;
           IDE_OE1 : out  STD_LOGIC;
           IDE_OE2 : out  STD_LOGIC;
           IDE_OE3 : out  STD_LOGIC;
           IDE_CLK2 : out  STD_LOGIC;
           IDE_BDIR : out  STD_LOGIC;
              -- Main oscillator Clock, divided to get the required CPU clock
           MAINCLK : in  STD_LOGIC;
           ADR12 : in  STD_LOGIC;
           ADR13 : in  STD_LOGIC;
           ADR14 : in  STD_LOGIC;
           ADR2 : in  STD_LOGIC;
           ADR3 : in  STD_LOGIC;
              ADR4 : in  STD_LOGIC;
              -- UART 6850 chip sel, R/W and E lines
           UARTSEL : out  STD_LOGIC;
           UARTRW : out  STD_LOGIC;
           UARTE : out  STD_LOGIC;
              -- Baudrate 1.8432 Mhz clock in
           BAUDCLK : in  STD_LOGIC;
              -- Baud * 16 clock out
           BAUDCLK16 : out  STD_LOGIC;
              -- Ram/Rom control signals OE and WE
           ROMOE : out  STD_LOGIC;
           RAMOE : out  STD_LOGIC;
           RAMWE : out  STD_LOGIC;
              -- Cpu control signals
           MNEGIO : in  STD_LOGIC;
           WRP : in  STD_LOGIC;
```

```
            NEGR_W : in  STD_LOGIC;
            NOTINTREQ : out  STD_LOGIC;
            INTACK : in  STD_LOGIC;
            OPREQ : in  STD_LOGIC;
               -- Master Reset, coming from the pushbutton
            MRESET : in  STD_LOGIC;
            CPUCLK : out  STD_LOGIC;
               -- Cpu reset
            CPURESET : out  STD_LOGIC);
end SG2650_Logic;

architecture Behavioral of SG2650_Logic is

alias EnegNE is ADR13;
alias DnegC is ADR14;

signal theClock : STD_LOGIC;

-- when '0' we are in the first 8K of a MEMORY access
signal first4KMem : STD_LOGIC;

-- when '1' we are executing a 'write to data' instruction WRTD
signal writeToData : STD_LOGIC;

-- a bit to say if the first 8K are ROM '0' or RAM '1'
signal first4KBit : STD_LOGIC;

-- this is '0' when there's a Mem access to RAM
signal isTheRam : STD_LOGIC;

-- this is an important signal, basically is OPREQ AND CPUCLOCK, this
-- is guaranteed to be a good strobe to latch data/stuff for I/O
signal theStrobe : STD_LOGIC;

-- a duplication as we need it to combine for ramoe
signal theRomOE : STD_LOGIC;

-- delayed OPREQ
signal delayedOpreq : STD_LOGIC;

signal UartCS : STD_LOGIC;
signal UartIsSelected : STD_LOGIC;

signal IdeHISelected : STD_LOGIC;
signal IdeLowSelected : STD_LOGIC;

-- this is to generate the ide read/write signal
signal IdeStrobe : STD_LOGIC;

signal theIdeRD : STD_LOGIC;

signal theResetCounter : STD_LOGIC_VECTOR (2 downto 0);
signal ResCntNotZero : STD_LOGIC;

-- (7+1) x 2 = 16
signal theCpuClkDiv : STD_LOGIC_VECTOR (2 downto 0) := "111";

-- (5+1) x 2 = 12 so 1.8432 Mhz / 12 = 153.600 Khz = 9600 x 16
signal theBaudClkDiv : STD_LOGIC_VECTOR (2 downto 0) := "101";

component ClockDivider
    generic ( widthBits : integer := 2 );--default value is 2
```

```
    Port ( clkin : in STD_LOGIC;
           div : in STD_LOGIC_VECTOR (widthBits downto 0);
           clkout : out STD_LOGIC;
           negreset : in STD_LOGIC );
end component;

begin

-- the CPU clock we assume is MAINCLK divided by 4 ( to get 1 Mhz )

CpuClkDivide : ClockDivider generic map (widthBits => 2) port map ( MAINCLK,
                                                    theCpuClkDiv, theClock, MRESET );

CPUCLK <= theClock;

-- let's make a CPU reset that is guaranteed to stay for some clock cycles

ResCntNotZero <= OR_REDUCE ( theResetCounter ); -- or of all its bits

CPURESET <= ResCntNotZero; -- '1' as long as it's not 0

NOTINTREQ <= '1'; -- for now no interrupts

RESCPU: process ( MRESET, theClock )
         begin
          if ( MRESET = '0') then
               theResetCounter <= "101"; -- 5-1 = 4 clock cycles
           elsif ( theClock'event and theClock = '1') then
             if ( ResCntNotZero = '1' ) then
              theResetCounter <= theResetCounter-1;
             end if;
          end if;
         end process;

-- as explained before, this guarantes a valid 'strobe' when
-- all data and control lines are definitely valid
-- the fact is OPREQ goes high BEFORE the clock can do so this
-- gives us a bit of time for settling of the signals.
theStrobe <= theClock and OPREQ;

-- remember NOTHING is valid until OPREQ = '1'
first4KMem <= '0' when MNEGIO = '1' and ADR14 = '0' and ADR13 = '0' and ADR12 = '0'
                                    and OPREQ = '1' else '1';

-- ADR13 is ALSO EnegNE ( '1' = extended, '0' = NOT extended )
-- ADR4,ADR3 = "00" = map latch
writeToData <= '1' when MNEGIO = '0' AND theStrobe = '1' and EnegNE ='1'
                                     and ADR3 = '0' and ADR4 = '0' else '0';

-- fundamentally latch ADR2 on the rasing edge of that
-- this 'shit' because we do not have any data line available
-- to latch so we latch ADR2 instead
THEfirst4KBIT : process ( MRESET, writeToData )
                    begin
                     if ( MRESET = '0' ) then
                       first4KBit <= '0'; -- at reset it must be rom
                     elsif ( writeToData'event and writeToData = '1' ) then
                       first4KBit <= ADR2;
                     end if;
                    end process;

-- now as simple as possible the rom/ram OE/WE signals
```

```
ROMOE <= theRomOE;
-- this combination already includes MNEGIO and OPREQ
theRomOE <= '0' when first4Kbit = '0' and first4KMem = '0' and OPREQ = '1' else '1';

-- fundamentally MNEGIO '1' and OPREQ '1' == memory access of any kind
isTheRam <= '0' when theRomOE = '1' and MNEGIO = '1' and OPREQ = '1' else '1';

-- that's RAMOE
RAMOE <= isTheRam or NEGR_W;

-- and that's RAMWE
RAMWE <= isTheRam or (not (NEGR_W));

-- now let's sort out the 6580
-- we assume the CPU and the Uart both go the same 1Mhz clock

-- NOTE : this is the NOT of the CPU clock ( must be 1Mhz )
UARTE <= not ( theClock );

DELAYOPREQ: process ( theClock )
               begin
                if ( theClock'event and theClock='1') then
                       delayedOpreq <= OPREQ;
                end if;
               end process;

-- it works using a EXTENDED I/O instruction, ADR4,ADR3 = "01"
UartIsSelected <= '0' when EnegNE = '1' and MNEGIO = '0' and ADR3 = '1'
                                        and ADR4 = '0' else '1';
UartCS <= not ((delayedOpreq and OPREQ)) or UartIsSelected;
UARTRW <= not (NEGR_W); -- neg because on the 6850 is RnegW i.e. '0' = write, '1' = read
UARTSEL <= UartCS;

BaudClkDivide : ClockDivider generic map (widthBits => 2) port map ( BAUDCLK, theBaudClkDiv,
                                                                      BAUDCLK16, MRESET );

-- tricky part, time to check about those IDE signals
-- so we have ADR4,ADR3 = "10" IDE_L and ADR4,ADR3 = "11" IDE_H

IdeLowSelected <= '0' when EnegNE = '1' and MNEGIO = '0' and ADR3 = '0'
                                        and ADR4 = '1' else '1';
IdeHiSelected <= '0' when EnegNE = '1' and MNEGIO = '0' and ADR3 = '1'
                                        and ADR4 = '1' else '1';

THEIDESTB : process ( MRESET, theClock )
             begin
              if ( MRESET = '0' ) then
                 IdeStrobe <= '1';
               elsif ( theClock'event and theClock = '1') then
                if ( OPREQ = '1' ) then
                 IdeStrobe <= not ( IdeStrobe );
                end if;
               end if;
             end process;

theIdeRD <= IdeStrobe or NEGR_W or IdeLowSelected;
IDE_RD <= theIdeRD;

IDE_WR <= IdeStrobe or not (NEGR_W) or IdeLowSelected;
IDE_CS0 <= not ( OPREQ ) or IdeLowSelected;

IDE_OE3 <= IdeLowSelected or not ( OPREQ );
```

```
IDE_BDIR <= NEGR_W; -- 0 = read = A <- B, 1 = write A -> B

-- high 'read' latch, when I read HI
IDE_OE1 <= IdeHiSelected or NEGR_W or not ( OPREQ );

-- high 'write' latch OUTPUT, when I write on LOW ( which does the IDE_CS cycle )
IDE_OE2 <= IdeLowSelected or not ( OPREQ ) or not ( NEGR_W );

-- high 'write' latch STORE OUTPUT, when I write on high
IDE_CLK2 <= not (IdeStrobe) or IdeHiSelected or not (NEGR_W);

-- when you read 'low' it also latches 'high'
IDE_CLK1 <= theIdeRD; -- remember it latches on the RAISING EDGE

end Behavioral;
```

6.1 The UCF file for the CPLD pins assignement

```
#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "ADR12"  LOC = "P6"  ;
NET "ADR13"  LOC = "P28"  ;
NET "ADR14"  LOC = "P29"  ;
NET "ADR2"  LOC = "P7"  ;
NET "ADR3"  LOC = "P40"  ;
NET "ADR4"  LOC = "P42"  ;
NET "BAUDCLK"  LOC = "P27"  ;
NET "BAUDCLK16"  LOC = "P24"  ;
NET "CPUCLK"  LOC = "P43"  ;
NET "CPURESET"  LOC = "P44"  ;
NET "IDE_BDIR"  LOC = "P13"  ;
NET "IDE_CLK1"  LOC = "P4"  ;
NET "IDE_CLK2"  LOC = "P9"  ;
NET "IDE_CS0"  LOC = "P3"  ;
NET "IDE_OE1"  LOC = "P8"  ;
NET "IDE_OE2"  LOC = "P11"  ;
NET "IDE_OE3"  LOC = "P12"  ;
NET "IDE_RD"  LOC = "P1"  ;
NET "IDE_WR"  LOC = "P2"  ;
NET "INTACK"  LOC = "P37"  ;
NET "MAINCLK"  LOC = "P5"  ;
NET "MNEGIO"  LOC = "P33"  ;
NET "MRESET"  LOC = "P39"  ;
NET "NEGR_W"  LOC = "P35"  ;
NET "NOTINTREQ"  LOC = "P36"  ;
NET "OPREQ"  LOC = "P38"  ;
NET "RAMOE"  LOC = "P19"  ;
NET "RAMWE"  LOC = "P22"  ;
NET "ROMOE"  LOC = "P20"  ;
NET "UARTE"  LOC = "P26"  ;
NET "UARTRW"  LOC = "P18"  ;
NET "UARTSEL"  LOC = "P14"  ;
NET "WRP"  LOC = "P34"  ;

#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE
```

7 Card monitor/bootloader program

The card monitor/booloader program is resident in a 4K Eprom, it allows initilisation of memory and UART and contains a mini monitor program that allows to :

- Display contents of memory
- Modify memory contents
- Load a program into memory via XMODEM
- Execute code

The monitor is accessed by connecting a computer terminal via serial port set at 9600 Baud, 8 bits, no parity . At power up a welcome message is displayed such as :

```
Signetics 2650 CPU Board @1Mhz
(C) 2012 Ivan Z. Llamasoft
BootLoader Version 5.02
>
```

Any time the prompt '>' is displayed the monitor is ready to accept commands. Commands are a single letter eventually followed by two or four hexadecimal digits.

7.1 Monitor command 'H', Help

The monitor command 'H' displays a help text showing the list of available commans.

```
>h
H shows help, D <aaaa> dumps memory
A <aaaa> alters memory ( '.' to exit )
L <aaaa> XMODEM loads data, ^X CAN
J <aaaa> jumps to address
<aaaa> hexadecimal 16 bits address
>
```

7.2 Monitor command 'D' <aaaa>, Dump memory

The monitor command 'D' displays an hexadecimal and ASCII dump of a 256 bytes memory block starting from address 'aaaa' (address can be from 0000H to 7FF00H) , the non printable ASCII characters are replaced by dots.

```
>d 0000
0000  20 93 C0 C0 C0 C0 C0 C0 05 13 D5 08 C0 C0 C0 C0    ................
0010  C0 C0 D5 08 C0 C0 C0 C0 05 11 D5 08 C0 C0 04 04   ................
0020  CC 10 00 04 C1 CC 10 01 3F 02 5F 04 3E 3F 02 8F   ........?._.>?..
0030  3F 01 FB 04 05 CC 10 00 04 16 CC 10 01 3F 02 5F   ?............?._
0040  05 00 3F 01 F0 60 18 63 85 01 CD 10 03 E4 61 1A   ..?..`.c......a.
0050  02 A4 20 E4 48 98 10 04 05 CC 10 00 04 19 CC 10   .. .H...........
0060  01 3F 02 5F 1F 00 2B E4 44 9C 01 2B 0D 10 03 3F   .?._..+.D..+...?
0070  01 F0 60 1C 00 2B 3F 02 A1 CC 10 05 CC 10 0E 3F   ..`..+?........?
0080  02 A1 CC 10 06 CC 10 0F 05 10 CD 10 07 05 10 CD   ................
0090  10 08 0C 10 05 3F 02 D4 0C 10 06 3F 02 D4 04 20   .....?.....?...
00A0  3F 02 8F 3F 02 8F 0C 90 05 3F 02 D4 04 20 3F 02   ?..?.....?... ?.
00B0  8F 04 01 05 01 8D 10 06 18 02 04 00 CD 10 06 8C   ................
00C0  10 05 CC 10 05 0C 10 08 A4 01 CC 10 08 98 57 04   ..............W.
00D0  20 3F 02 8F 3F 02 8F 05 10 CD 10 08 0C 90 0E E4    ?..?...........
00E0  20 1A 26 E4 7F 19 22 3F 02 8F 0C 10 0F 84 01 CC    .&.."?........
00F0  10 0F 98 08 0C 10 0E 84 01 CC 10 0E 0C 10 08 A4   ................
>
```

7.3 Monitor command 'A' <aaaa>, Alter memory contents

The monitor command 'A' allows to modify memory contents starting from address 'aaaa' (address can be from 0000H to 7FFFFH), it enters an interactive mode where the address and the current content are shown, by entering a two digit hexadecimal number you can modify the content. Entering a DOT character exits and returns to the commands prompt terminating the modify session.

```
>a 2000
2000  00 ? 1
2001  00 ? 23
2002  00 ? 55
2003  00 ? .
>
```

7.4 Monitor command 'L' <aaaa>, Load data at address (via XMODEM)

The monitor command 'L' allows to load data into memory starting ad adress 'aaaa' (address can be from 0000H to 7FFFFH) using the XMODEM protocol (simple CRC, 128 bytes packet size). This allows to load code or data into a memory address for further execution. When the loading is finished the monitor returns to the command prompt.

Transfer can be aborted in any moment by sending two consecutive CAN characters (CTRL+X) as by XMODEM protocol.

```
>l 2000
Transfer aborted.
>
```

7.5 Monitor used memory

The monitor starts at address 0000H in ROM and uses some memory locations from 1000H to 1040H about to hold some variables and states. Please refer to the monitor listing for more details.

7.6 Monitor listing

Here is a complete 2650 assembler listing of the bootloader/monitor program.

```
        tabs    10,8        ;first tab is 10, then each tab is 8 ahead
        width   132         ;prevent folding of long lines
        noproces            ;disable fancy stuff, speed is what we want
        ;
        title   "Bootloader␣1"
        stitle  "Written␣by␣Ivan␣Z.␣Llamasoft␣2012"
        ;
        name    boot1_2650  ;the module name
        ;

        org     0           ; ROM start address

lf      equ     10          ; line feed
cr      equ     13          ; carriage return
bksp    equ      8          ; backspace
bell    equ      7          ; the bell char
```

```
del      equ        127         ; delete char

         ;name the condition codes,
         ;(we don't remember numbers anyway)

Z        equ       0    ; zero
EQ       equ       z
P        equ       1
GT       equ       p
N        equ       2
LT       equ       n
UN       equ       3    ; always true

        ; name the registers

;R0     equ     0
;R1     equ     1
;R2     equ     2
;R3     equ     3

        ; some values for I/O space

UARTC   equ 08h ; UART control register ( write )
UARTS   equ 08h ; UART status register ( read )
UARTTX  equ 09h ; UART transmit register ( write )
UARTRX  equ 09h ; UART receive register ( read )

CPLDROM equ 0       ; write to this, first 8K = ROM
CPLDRAM equ 8       ; write to this, first 8K = RAM

IDEHI   equ 20h ; IDE register High Byte
IDELOW  equ 30h ; IDE register Low Byte

res_uart    equ 13h ; no RX int, no TX int, 8 bits + 2 stop, Master Reset
uart_conf   equ 11h ; no RX int, no TX int, 8 bits + 2 stop, clock / 16

RDRF    equ 1           ; if '1' data is ready in the RX register
TDRE    equ 2           ; if '1' the TX register is empty

MsgHi   equ 1000h
MsgLo   equ 1001h       ; stores the address in memory where the message is
R0_save equ 1002h       ; place to save R0
R1_save equ 1003h       ; place to save R1
R2_save equ 1004h       ; place to save R2
MemHi   equ 1005h       ; place to save a memory address for dump/load/etc.
MemLo   equ 1006h
Cnt1    equ 1007h       ; a couple of counters
Cnt2    equ 1008h

StrBuf  equ 1020h       ; string buffer where readstring puts stuff in

;
; Definitions for the XMODEM loader
;
CHAR_STX    equ 02H
CHAR_SOH    equ 01H
CHAR_CAN    equ 18H
CHAR_ACK    equ 06H
CHAR_NAK    equ 15H
CHAR_EOT    equ 04H

; Variables for the Xmodem Loader
```

```
X_Phase     equ 1009h   ; xmodem loader phase of the state machine
X_CanCnt    equ 100ah   ; counter of how many CHAR_CAN we received
X_LastPk    equ 100bh   ; last packet received number
X_AdrHi     equ 100ch
X_AdrLo     equ 100dh   ; hi/low address of the packet we are receiving
X_CurAdrHi  equ 100eh   ; current packet address in memory Hi
X_CurAdrLo  equ 100fh   ; "" "" and Lo
X_Crc       equ 1010h   ; current CRC in computation
X_ByteCntH  equ 1011h   ; bytes counter Hi
X_ByteCntL  equ 1012h   ; bytes counter Low ( in reality up to 1024 + 3 )
X_TimeCntH  equ 1013h   ; timeout counter Hi
X_TimeCntL  equ 1014h   ; timeout counter Lo
X_CntTwo    equ 1015h   ; a counter for the first 2 packet bytes ( pkt num )
X_CurPknum  equ 1016h   ; current packet number, if == X_LastPk DO NOT update ptrs
X_OneK      equ 1017h   ; "1K␣flag", if not zero we are using 1K packets
PtrHi       equ 1018h
PtrLo       equ 1019h


start   eorz    R0          ; r0 = 0
        lpsl                ; load status low from R0

        ; Reset uart and all the other shit

        nop
        nop
        nop
        nop
        nop
        nop

        lodi,R1 res_uart
        wrte,R1 UARTC       ; master reset to the uart
        nop
        nop
        nop
        nop
        nop
        nop
        wrte,R1 UARTC       ; master reset to the uart
        nop
        nop
        nop
        nop
        lodi,R1 uart_conf
        wrte,R1 UARTC       ; configure the uart for 9600 baud 8 bits no parity 1 stop
        nop
        nop

        lodi,R0 hi(bootmsg)
        stra,R0 MsgHi
        lodi,R0 lo(bootmsg)
        stra,R0 MsgLo       ; save address of message

        bsta,UN WriteMsg    ; write the boot message

        ; main prompt loop

prompt:
        lodi,R0 '>'         ; the prompt
```

```
        bsta,UN WriteCh     ; write it

        ; let's wait util we get something from the KB
        bsta,UN GetStr

        ; print a CR/LF
        lodi,R0 hi(m_crlf)
        stra,R0 MsgHi
        lodi,R0 lo(m_crlf)
        stra,R0 MsgLo       ;
        bsta,UN WriteMsg    ; write CR/LF

        lodi,R1 0           ; start at the beginning of the string
        bsta,UN SkipWhite   ; skip the white spaces and get something in R0
        iorz R0
        bctr,EQ prompt      ; if it's an EQ we have an empty string

        ; if we are here R0 is the first not null char we may have something to do
        ; now this is a bit brutal but ..

        addi,R1 1           ; because for further things we want to be PAST this char
        stra,R1 R1_save     ; let's save R1 as index to the first non-white

        comi,R0 'a'         ; is it >= 'a'
        bctr,LT command     ; no, fine as it is
        subi,R0 32          ; yes, make it uppercase then 'A' .. 'Z'

command:
        ; time to interpret what we have
        comi,R0 'H'         ; is it 'help' ?
        bcfr,EQ no_help     ; no it,s not

        ; yes it's help
        lodi,R0 hi(m_help)
        stra,R0 MsgHi
        lodi,R0 lo(m_help)
        stra,R0 MsgLo       ;
        bsta,UN WriteMsg    ; write the help info
        bcta,UN prompt      ; go back to prompt

no_help:
        comi,R0 'D'         ; is dump ?
;       bcfr,EQ no_dump     ; no it,s not
        bcfa,EQ no_dump     ; no it,s not

        ; is dump, let's get two numbers
        ; This is DUMP <aaaa>

        loda,R1 R1_save     ; want that index back
        bsta,UN SkipWhite   ; skip the white spaces and get something in R0
;       bctr,EQ prompt      ; if it's an EQ we have an empty string
        iorz R0
        bcta,EQ prompt      ; if it's an EQ we have an empty string

        ; let's get a 16 bits number
        bsta,UN Get8Hex     ; first HIGH digit in R0
        stra,R0 MemHi
        stra,R0 X_CurAdrHi  ; a copy in here too
        bsta,UN Get8Hex     ; second LOW digit in R0
        stra,R0 MemLo
        stra,R0 X_CurAdrLo  ; idem
```

```
        ; time to dump all this
        lodi,R1 16
        stra,R1 Cnt1        ; 16 lines counter

l_loop  lodi,R1 16
        stra,R1 Cnt2        ; 16 bytes at time

        loda,R0 MemHi
        bsta,UN Write8Hex   ; write the high address byte
        loda,R0 MemLo
        bsta,UN Write8Hex   ; write the low address byte
        lodi,R0 ' '
        bsta,UN WriteCh     ; write a couple
        bsta,UN WriteCh     ; of spaces

b_loop  loda,R0 *MemHi      ; R0 = *(AdrHi-AdrLo)
        bsta,UN Write8Hex   ; write the byte at that mem loc
        lodi,R0 ' '
        bsta,UN WriteCh     ; write a space

        lodi,R0 1           ; assume 1 for MemHI
        lodi,R1 1           ; 1 for MemLo in ANY case
        adda,R1 MemLo       ; R1 = 1 + *(MemLo)
        bctr,EQ inc_hm      ; if 0 means Hi has to be incremented too
        lodi,R0 0           ; else leave MemHi untouched
inc_hm  stra,R1 MemLo       ; save again MemLo
        adda,R0 MemHi       ; R0 = 1/0 + *(MemHi)
        stra,R0 MemHi

        loda,R0 Cnt2        ; decrement the row counter
        subi,R0 1
        stra,R0 Cnt2
        bcfr,EQ b_loop      ; go in loop if not zero

        ; 1 line of 16 bytes been dumped

        ; new stuff, the ASCII dump

        lodi,R0 ' '
        bsta,UN WriteCh     ; write a space
        bsta,UN WriteCh     ; write another space

        lodi,R1 16
        stra,R1 Cnt2        ; 16 bytes again at time

asc_lp  loda,r0 *X_CurAdrHi
        comi,R0 32          ; is something between space and 127 ?
        bctr,LT no_asc      ; is < 32, ignore it
        comi,R0 127
        bctr,GT no_asc      ; is > 127, ignore it
        bsta,UN WriteCh     ; write the char

sublp:  loda,R0 X_CurAdrLo
        addi,R0 1
        stra,R0 X_CurAdrLo
        bcfr,EQ noincha
        loda,R0 X_CurAdrHi
        addi,R0 1
        stra,R0 X_CurAdrHi

noincha loda,R0 Cnt2        ; decrement the ascii bytes counter
        subi,R0 1
```

```
        stra,R0 Cnt2
        bcfr,EQ asc_lp      ; go in loop if not zero
        bsta,UN dolf        ; finally go next line

no_asc  lodi,R0 '.'
        bsta,UN WriteCh     ; write a dot
        bctr,UN sublp       ; and go in loop again

        ; print a CR/LF
dolf    lodi,R0 hi(m_crlf)
        stra,R0 MsgHi
        lodi,R0 lo(m_crlf)
        stra,R0 MsgLo       ;
        bsta,UN WriteMsg    ; write CR/LF

        loda,R0 Cnt1        ; decrement the line counter
        subi,R0 1
        stra,R0 Cnt1
;       bcfr,EQ l_loop      ; go in loop if not zero
        bcfa,EQ l_loop      ; go in loop if not zero

        bcta,UN prompt      ; end of dump


no_dump:
        comi,R0 'A'         ; is it 'alter' ?
;       bcfr,EQ no_alter        ; no it,s not
        bcfa,EQ no_alter        ; no it,s not

        ; is alter, let's get two numbers

        loda,R1 R1_save     ; want that index back
        bsta,UN SkipWhite   ; skip the white spaces and get something in R0
;       bctr,EQ prompt      ; if it's an EQ we have an empty string
        bcta,EQ prompt      ; if it's an EQ we have an empty string

        ; let's get a 16 bits number
        bsta,UN Get8Hex     ; first HIGH digit in R0
        stra,R0 MemHi
        bsta,UN Get8Hex     ; second LOW digit in R0
        stra,R0 MemLo

a_loop:
        loda,R0 MemHi
        bsta,UN Write8Hex   ; write the high address byte
        loda,R0 MemLo
        bsta,UN Write8Hex   ; write the low address byte
        lodi,R0 ' '
        bsta,UN WriteCh     ; write a couple
        bsta,UN WriteCh     ; of spaces

        loda,R0 *MemHi      ; R0 = *(AdrHi-AdrLo)
        bsta,UN Write8Hex   ; write the byte at that mem loc
        lodi,R0 ' '
        bsta,UN WriteCh     ; write a space
        lodi,R0 '?'
        bsta,UN WriteCh     ; write a question mark
        lodi,R0 ' '
        bsta,UN WriteCh     ; write a space

        ; let's wait util we get something from the KB
        bsta,UN GetStr
```

```
        ; print a CR/LF
        lodi,R0 hi(m_crlf)
        stra,R0 MsgHi
        lodi,R0 lo(m_crlf)
        stra,R0 MsgLo       ;
        bsta,UN WriteMsg    ; write CR/LF

        lodi,R1 0           ; start at the beginning of the string
        bsta,UN SkipWhite   ; skip the white spaces and get something in R0
        bctr,EQ a_loop  ; if it's an EQ we have an empty string

        comi,R0 '.'
;       bctr,EQ prompt      ; if it's a '.' end here
        bcta,EQ prompt      ; if it's a '.' end here

        ; it's not a '.' we assume is a valid digit
        bsta,UN Get8Hex     ; get digit in R0
        stra,R0 *MemHi      ; write it into memory

        lodi,R0 1           ; assume 1 for MemHI
        lodi,R1 1           ; 1 for MemLo in ANY case
        adda,R1 MemLo       ; R1 = 1 + *(MemLo)
        bctr,EQ inc_hm2     ; if 0 means Hi has to be incremented too
        lodi,R0 0           ; else leave MemHi untouched
inc_hm2 stra,R1 MemLo       ; save again MemLo
        adda,R0 MemHi       ; R0 = 1/0 + *(MemHi)
        stra,R0 MemHi

;       bctr,UN a_loop      ; continue until '.'
        bcta,UN a_loop      ; continue until '.'

no_alter:
        comi,R0 'L'         ; is it 'load' ?
        bcfa,EQ no_load     ; no it,s not

        lodi,R0 11h         ;
        stra,R0 X_AdrHi
        lodi,R0 00h
        stra,R0 X_AdrLo     ; assume 1100h as default load address

        loda,R1 R1_save     ; want that index back
        bsta,UN SkipWhite   ; skip the white spaces and get something in R0
        bctr,EQ go_xmo      ; if it's an EQ we have an empty string

        ; let's get a 16 bits number
        bsta,UN Get8Hex     ; first HIGH digit in R0
        stra,R0 X_AdrHi
        bsta,UN Get8Hex     ; second LOW digit in R0
        stra,R0 X_AdrLo

go_xmo: bcta,UN Do_Xmdm     ; go, do the XMODEM protocol
        bcta,UN prompt      ; safety ...
no_load:
        comi,R0 'J'         ; is it 'jump' ?
        bcfa,EQ no_jump     ; no it,s not

        loda,R1 R1_save     ; want that index back
        bsta,UN SkipWhite   ; skip the white spaces and get something in R0
        bcta,EQ prompt      ; if empty, no jump

        ; let's get a 16 bits number
```

```
        bsta,UN Get8Hex     ; first HIGH digit in R0
        stra,R0 MemHi
        bsta,UN Get8Hex     ; second LOW digit in R0
        stra,R0 MemLo
        bcta,UN *MemHi          ; **JUMP ! ** Ta ta ta ta tata ta ..
no_jump:
end_cmd:
        ; all unrecognized commands end up here
        bcta,UN prompt

        ; Skips the white spaces in a string starting at index = R1
        ; returns R1 to the first non-white char or NULL

SkipWhite:
        subi,R1 1           ; --R1 cause the ++ immediately after this

skp_lp  loda,R0 StrBuf,R1+  ; get the char in R0
        retc,EQ             ; if we met a NULL we can end here
        comi,R0 ' '         ; is it a space ?
        bctr,EQ skp_lp      ; yes, move on
        retc,UN             ; no, we found something

        ; Gets a string into StrBuf, no longer than MAX_CHARS

GetStr  lodi,R2 0ffh        ; -1 cause the ++, this is our pointer

k_loop  rede,R1 UARTS       ; let's check the UART status
        andi,R1 RDRF        ; any char in the RX buffer ?
        bctr,EQ k_loop      ; if not just wait again

        ; we have some char
        rede,R0 UARTRX
        comi,R0 cr              ; is a carriage return ?
;       bctr,EQ endstr      ; yes we have something
        bcta,EQ endstr      ; yes we have something
        comi,R0 lf              ; is a line feed ?
;       bctr,EQ endstr      ; yes we have something
        bcta,EQ endstr      ; yes we have something
        comi,R0 bksp
        bctr,EQ backspace   ; go handle the backspace
        comi,R0 del         ; if it's a DEL ( 127 ) char
        bctr,EQ backspace   ; go handle the backspace
        comi,R0 32              ; is something between space and 127 ?
        bctr,LT k_loop      ; is < 32, ignore it
        comi,R0 127
        bctr,GT k_loop      ; is > 127, ignore it

        ; the char is good, can we put it in ?
        comi,R2 16
        bctr,EQ errbell     ; no, we are already full
        stra,R0 StrBuf,R2+  ; ++R2 and save it into the buffer
        stra,R0 R0_save
        lodi,R0 0
        stra,R0 StrBuf,R2+  ; ++R2 and add a null
        subi,R2 1           ; dec R2 so it's ready for the next char

        ; finally echo the char
        loda,R0 R0_save

echo_ch bsta,UN WriteCh     ; write it
        bctr,UN k_loop      ; get a new char
```

```
backspace:
        comi,R2 0ffh        ; are we at the first char ?
        bctr,EQ errbell     ; if so, bell
        lodi,R0 0
        stra,R0 StrBuf,R2   ; put a 0 at the current position
        subi,R2 1           ; R2--, go back 1 char
        lodi,R0 bksp
        bsta,UN WriteCh     ; write backspace
        lodi,R0 ' '
        bsta,UN WriteCh     ; write space over it
        lodi,R0 bksp
        bsta,UN WriteCh     ; write backspace
;       bctr,UN k_loop      ; go to get more chars
        bcta,UN k_loop      ; go to get more chars

        ; sound the bell if something is wrong
errbell:
        lodi,R0 bell        ; put the bell char in
        bctr,UN echo_ch     ; sound it and go back in loop

        ; we still have to add a NULL, in case one just presses enter

endstr  lodi,R0 0
        stra,R0 StrBuf,R2+  ; ++R2 and add a null
        retc,UN             ; unconditional return

        ; Writes a message NULL (0) terminated where
        ; absolute 15 bits address is in (MsgHi - MsgLo)

WriteMsg stra,R0 R0_save    ; save R0
         stra,R1 R1_save    ; save R1

msg_lp  loda,R0 *MsgHi      ; R0 = *(AdrHi-AdrLo)
        bctr,EQ endmsg      ; if 0 end of string

w_loop  rede,R1 UARTS       ; let be sure we can transmit first
        andi,R1 TDRE        ; check the transmitter empty bit
        bctr,EQ w_loop      ; if zero, wait until it gets '1'

        ; now we can transmit the byte

        wrte,R0 UARTTX      ; send the character

        ; let's increment address now
        lodi,R0 1           ; assume 1 for AdrHI
        lodi,R1 1           ; 1 for AdrLo in ANY case
        adda,R1 MsgLo       ; R1 = 1 + *(MsgLo)
        bctr,EQ inc_h       ; if 0 means Hi has to be incremented too
        lodi,R0 0           ; else leave MsgHi untouched
inc_h   stra,R1 MsgLo       ; save again MsgLo
        adda,R0 MsgHi       ; R0 = 1/0 + *(MsgHi)
        stra,R0 MsgHi

        bctr,UN msg_lp      ; continue with the message

        ; end of transmission, restore and return

endmsg  loda,R0 R0_save
        loda,R1 R1_save
        retc,UN             ; unconditional return

        ; Writes the char contained in R0
```

```
WriteCh stra,R0 R0_save
w_loop2 rede,R0 UARTS       ; let be sure we can transmit first
        andi,R0 TDRE        ; check the transmitter empty bit
        bctr,EQ w_loop2     ; if zero, wait until it gets '1'

        ; now we can transmit the byte
        loda,R0 R0_save     ; get it back
        wrte,R0 UARTTX      ; send the character

        loda,R0 R0_save
        retc,UN             ; unconditional return

        ; get 8 bits value from hex
        ; R1 = index to string buffer
        ; R0 = exit result
Get8Hex:
        subi,R1 1           ; -1 for index cause ++ now
        loda,R0 StrBuf,R1+  ; get the char in R2
        retc,EQ             ; if zero end
        subi,R0 '0'         ; we do it a bit simpler way
        comi,R0 10
        bctr,LT dig1ok      ; if < 10 we are ok
        subi,R0 7           ; adjust for "A"-"F"
        comi,R0 16
        bctr,LT dig1ok      ; if < 16 we are ok
        subi,R0 32          ; adjust for "a"-"f"
dig1ok  rrl,R0
        rrl,R0
        rrl,R0
        rrl,R0              ; put the high nibble where it should be
        stra,R0 R2_Save

        ; same story for the second digit

        loda,R0 StrBuf,R1+  ; get the char in R0
        retc,EQ             ; if zero end
        subi,R0 '0'         ; we do it a bit simpler way
        comi,R0 10
        bctr,LT dig2ok      ; if < 10 we are ok
        subi,R0 7           ; adjust for "A"-"F"
        comi,R0 16
        bctr,LT dig2ok      ; if < 16 we are ok
        subi,R0 32          ; adjust for "a"-"f"
dig2ok  iora,R0 R2_Save     ; low nibble in R0
        addi,R1 1           ; make it point to the char after this
        retc,UN             ; unconditional return

        ; Writes the 8 hex digit in R0

Write8Hex:
        stra,R0 R1_save
        loda,R2 R1_save
        rrr,R0
        rrr,R0
        rrr,R0
        rrr,R0
        andi,R0 0fH         ; get the high nibble and mask it
        addi,R0 '0'         ; sum this
        comi,R0 58          ; is the result < 58 ?
        bctr,LT dig1gd      ; if yes we are ok
        addi,R0 7           ; otherwise we need 7 more to be "A" "F"
```

```
dig1gd  bsta,UN WriteCh     ; write the high nibble

        ; lower nibble, same story
        loda,R0 R1_save
        andi,R0 0fH         ; get the low nibble and mask it
        addi,R0 '0'         ; sum this
        comi,R0 58          ; is the result < 58 ?
        bctr,LT dig2gd      ; if yes we are ok
        addi,R0 7           ; otherwise we need 7 more to be "A" "F"
dig2gd  bsta,UN WriteCh     ; write the low nibble
        retc,UN             ; unconditional return

        ; A subroutine to increment a 16 bits pointer
IncrPtr lodi,R0 1
AddPtr  adda,R0 *PtrLo      ; R0 = R0 + *(PtrHi)
        stra,R0 *PtrLo      ; save back LO + R0
        bctr,EQ pt_inch     ; we have an overflow of the "Lo" we can increment "Hi" too
        retc,UN             ; unconditional return
pt_inch lodi,R0 1
        adda,R0 *PtrHi
        stra,R0 *PtrHi
        retc,UN             ; unconditional return


;       lodi,R0 1
;       adda,R0 xxxL
;       stra,r0 xxxL
;       bcfr,EQ done1
;       lodi,R0 1
;       adda,r0 xxxH
;       stra,r0 xxxH
;done1:

        ; Xmodem Loader

        ; phase 0, packet still to begin, wait for SOH or STX and/or CAN
        ; phase 1, stx/soh got, getting pktnum, 255-pktnum
        ; phase 2, got those above now just getting data+crc while computing CRC


Do_Xmdm:        ; assuming somewhere the load address been put into AdrHi,AdrLo

        lodi,R0 0
        stra,R0 X_Phase     ; phase 0
        stra,R0 X_OneK      ; assume we are going for 128 bytes
        stra,R0 X_TimeCntH  ; timeout counter, force a timeout
        lodi,R0 1
        stra,R0 X_TimeCntL
        lodi,R0 0ffh
        stra,R0 X_LastPk    ; last packet number = 0xff
        lodi,R0 2
        stra,R0 X_CanCnt    ; CAN counter

        ; let's check the timeout counter
tim_check:
        loda,R0 X_TimeCntL
        subi,R0 1
        stra,R0 X_TimeCntL  ; X_TimeCntl--
        comi,R0 0ffh
        bcfr,EQ nosubh      ; if not negative go on
        loda,R0 X_TimeCntH  ; otherwise
        subi,R0 1           ; subtract 1
```

```
        stra,R0 X_TimeCntH  ; from the HI part too and save it

nosubh  loda,R0 X_TimeCntH
        iora,R0 X_TimeCntL
        bcfr,EQ no_timeout  ; if H OR L != 0 NO timeout

        ; we have a timeout

timeout lodi,R0 0
        stra,R0 X_Phase     ; we restart from phase 0
        stra,R0 X_Crc

trash   rede,R1 UARTS       ; let's check the UART status
        andi,R1 RDRF        ; any char in the RX buffer ?
        bctr,EQ end_tr      ; if not stop thrashing
        rede,R0 UARTRX      ; read and thrash chars
        nop
        nop
        nop                 ; waste a bit of time
        bctr,UN trash       ; continue to trash

end_tr  lodi,R0 CHAR_NAK
        bsta,UN WriteCh     ; send a NAK

        lodi,R0 0ffh        ; restore timeout counter
        stra,R0 X_TimeCntL
        stra,R0 X_TimeCntH

        bsta,UN XM_ResetPtr ; reset pointers and fall back here

no_timeout:
        rede,R1 UARTS       ; let's check the UART status
        andi,R1 RDRF        ; any char in the RX buffer ?

        bcta,EQ tim_check   ; if not continue to check for timeout

        ; here we got some char from the serial port

char_got:
        lodi,R0 0ffh
        stra,R0 X_TimeCntL
        stra,R0 X_TimeCntH  ; reset the timeout counter to MAX

        ; now we process the char depending on the phase

        loda,R0 X_Phase
        bcfa,EQ x_phase1    ; if it's not 0, could be phase1

x_phase0:

        ; here we are in phase 0, waiting for SOH or STX

        rede,R0 UARTRX      ; get what we got
        comi,R0 CHAR_CAN    ; is it a CAN character maybe ?
        bctr,EQ is_can      ; if yes go to process it

        ; it's not a CAN could be STX or SOH

        comi,R0 CHAR_SOH    ; is it a SOH ?
;       bctf,EQ maybe_stx   ; no, maybe it's an STX then
        bctr,EQ is_soh      ; yes go there
        comi,R0 CHAR_STX    ; is it an STX maybe ?
```

```
        bcfr,EQ maybe_eot   ; no, could be an EOT then

        ; here is an STX, it means there are 1024 bytes of data
        lodi,R0 04h
        stra,R0 X_ByteCntH
        lodi,R0 0
        stra,R0 X_ByteCntL  ; set for 1024 bytes ( 400h )
        bctr,UN go_one      ; and go in phase 1

maybe_eot:
        comi,R0 CHAR_EOT    ; is an EOT char ?
        bcta,EQ end_xloader ; if yes, end of the file transfer

trash1: bcta,UN trash       ; unrecognized shit, thrash all and send NAK

        ; it's a SOH, let's reset pointers, counters and CRC
is_soh:
        bsta,UN XM_ResetPtr ; reset pointers

go_one:
        lodi,R0 1
        stra,R0 X_Phase     ; phase = 1 now
        lodi,R1 2           ; two bytes in here ( pkt num and 255-pktnum ) we have to get
        stra,R1 X_CntTwo
        bcta,UN no_timeout  ; continue to read chars

is_can: loda,R0 X_CanCnt    ; we got a CAN, let' see how many
        subi,R0 1           ; CAN counter--
        stra,R0 X_CanCnt
        bcfa,EQ no_timeout  ; if it's not zero, let's read more

x_abort rede,R1 UARTS       ; let's check the UART status
        andi,R1 RDRF        ; any char in the RX buffer ?
        bctr,EQ end_tr2     ; if not stop thrashing
        rede,R0 UARTRX      ; read and thrash chars
        nop
        nop
        nop                 ; waste a bit of time
        bctr,UN x_abort     ; continue to trash

end_tr2 lodi,R0 hi(abortmsg)
        stra,R0 MsgHi
        lodi,R0 lo(abortmsg)
        stra,R0 MsgLo       ; save address of message
        bsta,UN WriteMsg    ; write the abort message

        bcta,UN prompt      ; go back to prompt, exit xmodem loader

x_phase1:
        comi,R0 1
        bcfr,EQ x_phase2    ; if it's not 1, could be 2

        ; here in phase 1 we have to get : 2 bytes ( pktnum and 255-pktnum )
        ;   ,   <data> , <crc>
        loda,R0 X_CntTwo
        subi,R0 1
        stra,R0 X_CntTwo
        bctr,EQ got_num     ; we got both bytes

        rede,R0 UARTRX      ; get what we got
        stra,R0 X_CurPknum  ; save it temporary here ( i.e. packet[0], it will
                            ; get overwritten )
```

```
        bcta,UN no_timeout  ; go in loop getting more bytes

got_num rede,R0 UARTRX      ; get what we got ( i.e. 255-pktnum )
        eora,R0 X_CurPknum  ; pkt_num ^ (255-pktnum)
        comi,R0 0ffh        ; if they are correct the result has to be 0xff
        bcfa,EQ trash1      ; if it's not 255 something is corrupted/wrong

go_two  lodi,R0 2           ; we can go in phase 2 now
        stra,R0 X_Phase
        bcta,UN no_timeout  ; go in loop getting more bytes

        ; phase 2, we are getting bytes now and computing the CRC
x_phase2:

        loda,R0 X_ByteCntH  ; is hi counter 0
        bcfr,EQ no_crc      ; no, can't be the CRC
        loda,R0 X_ByteCntL
;       is HI = LO == 0 ?
;       comi,R0 1           ; is HI == 0 and LO == 1 ?
        bcfr,EQ no_crc      ; no, can't be the CRC

        ; is the CRC !
        rede,R1 UARTRX      ; get what we got
        loda,R0 X_Crc       ; get the computed CRC
        comz R1             ; check if they match
        bcfa,EQ trash       ; if they don't trash everyting ( and send NAK )

pkt_good:
        ; here the packet is good and the CRC too, we can advance pointers
        ; if and only if the packet number is NOT the same of before

        loda,R0 X_CurPkNum
        coma,R0 X_LastPk
        bctr,EQ no_updateptr    ; if the numbers are the same don't update the PTRs

        stra,R0 X_LastPk    ; make LastPk = CurPkNum

        loda,R0 X_CurAdrHi
        stra,R0 X_AdrHi
        loda,R0 X_CurAdrLo
        stra,R0 X_AdrLo     ; advance pointers


no_updateptr:
        lodi,R0 0
        stra,R0 X_Phase     ; restart from phase 0
        stra,R0 X_Crc       ; reset CRC

        bsta,UN XM_ResetPtr ; put counters and stuff back

        lodi,R0 CHAR_ACK
        bsta,UN WriteCh     ; send a ACK

        bcta,UN no_timeout  ; go in loop getting chars

no_crc: ; we are reading/saving bytes and computing the CRC on them then

        rede,R0 UARTRX      ; get what we got
        stra,R0 *X_CurAdrHi     ; save the byte
        adda,R0 X_Crc       ; compute the CRC
        stra,R0 X_Crc       ; and update it
```

```
        ; time to increment the pointer
        loda,R0 X_CurAdrLo
        addi,R0 1           ; increment the low part
        stra,R0 X_CurAdrLo  ; and save it
        bcfr,EQ no_inchi    ; if <> 0 no need to increment hi
        loda,R0 X_CurAdrHi
        addi,R0 1           ; increment the high part
        stra,R0 X_CurAdrHi
no_inchi:
        ; we need to decrement the bytes counter
        loda,R0 X_ByteCntL
        subi,R0 1
        stra,R0 X_ByteCntL  ;
        comi,R0 0ffh
        bcfr,EQ nosubh2     ; if not negative go on
        loda,R0 X_ByteCntH  ; otherwise
        subi,R0 1           ; subtract 1
        stra,R0 X_ByteCntH  ; from the HI part too and save it
nosubh2:

        ; this should never be 0, if it gets 0 here there's something weird
        bcta,UN no_timeout  ; go in loop getting chars

        ; end of all xloader, in a good way

end_xloader:

        lodi,R0 CHAR_ACK
        bsta,UN WriteCh     ; send a ACK

        ; waste a bit of time and trash everything that comes

        lodi,R0 020h
waste   lodi,R1 0ffh
waste1  nop
        nop
        nop
        nop
no_rd   subi,R1 1
        bcfr,EQ waste1      ; waste a bit of time
        subi,R0 1
        bcfr,EQ waste

        lodi,R0 hi(loadok)
        stra,R0 MsgHi
        lodi,R0 lo(loadok)
        stra,R0 MsgLo       ; save address of message
        bsta,UN WriteMsg    ; write the boot message

        bcta,UN prompt      ; end go back to prompt


; Mini routine, resets PTRs and counters
;
; let's rememebr a packet is <soh/stx><blk><255-blk><data>[128 or 1024]<crc>
; <crc> = sum all data ONLY

XM_ResetPtr:
        lodi,R0 0
        stra,R0 X_Crc       ; reset CRC
        stra,R0 X_ByteCntL
        stra,R0 X_ByteCntH  ; high bytes counter as well
```

```
        loda,R0 X_OneK
        bctr,EQ is_128      ; we are using 128 bytes
        lodi,R0 04h
        stra,R0 X_ByteCntH  ; otherwise is 400h, 1K bytes
        bctr,UN set_ptr     ; go on with the rest
is_128  lodi,R0 128         ; otherwise just set the low counter to 128
        stra,R0 X_ByteCntL  ; low bytes counter too

set_ptr loda,R0 X_AdrHi
        stra,R0 X_CurAdrHi
        loda,R0 X_AdrLo
        stra,R0 X_CurAdrLo  ; X_CurAdr = X_Adr
        retc,UN             ; unconditional return

; The various messages

bootmsg db cr,lf
        db "Signetics␣2650␣CPU␣Board␣@1Mhz"
        db cr,lf
        db "(C)␣2012␣Ivan␣Z.␣Llamasoft"
        db cr,lf
        db "BootLoader␣Version␣5.02"
m_crlf  db cr,lf
        db 0
m_help  db "H␣shows␣help,␣D␣<aaaa>␣dumps␣memory", cr, lf
        db "A␣<aaaa>␣alters␣memory␣(␣'.'␣to␣exit␣)",cr,lf
        db "L␣<aaaa>␣XMODEM␣loads␣data,␣^X␣CAN",cr,lf
        db "J␣<aaaa>␣jumps␣to␣address",cr,lf
        db "<aaaa>␣hexadecimal␣16␣bits␣address",cr,lf
        db 0
abortmsg:
        db  "Transfer␣aborted.",cr,lf
        db 0
loadok  db cr,lf,"Data␣loaded␣OK",cr,lf
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

8 Card pictures

Here pictures of the completed card showing jumpers locations.

Figure 9: The board components side

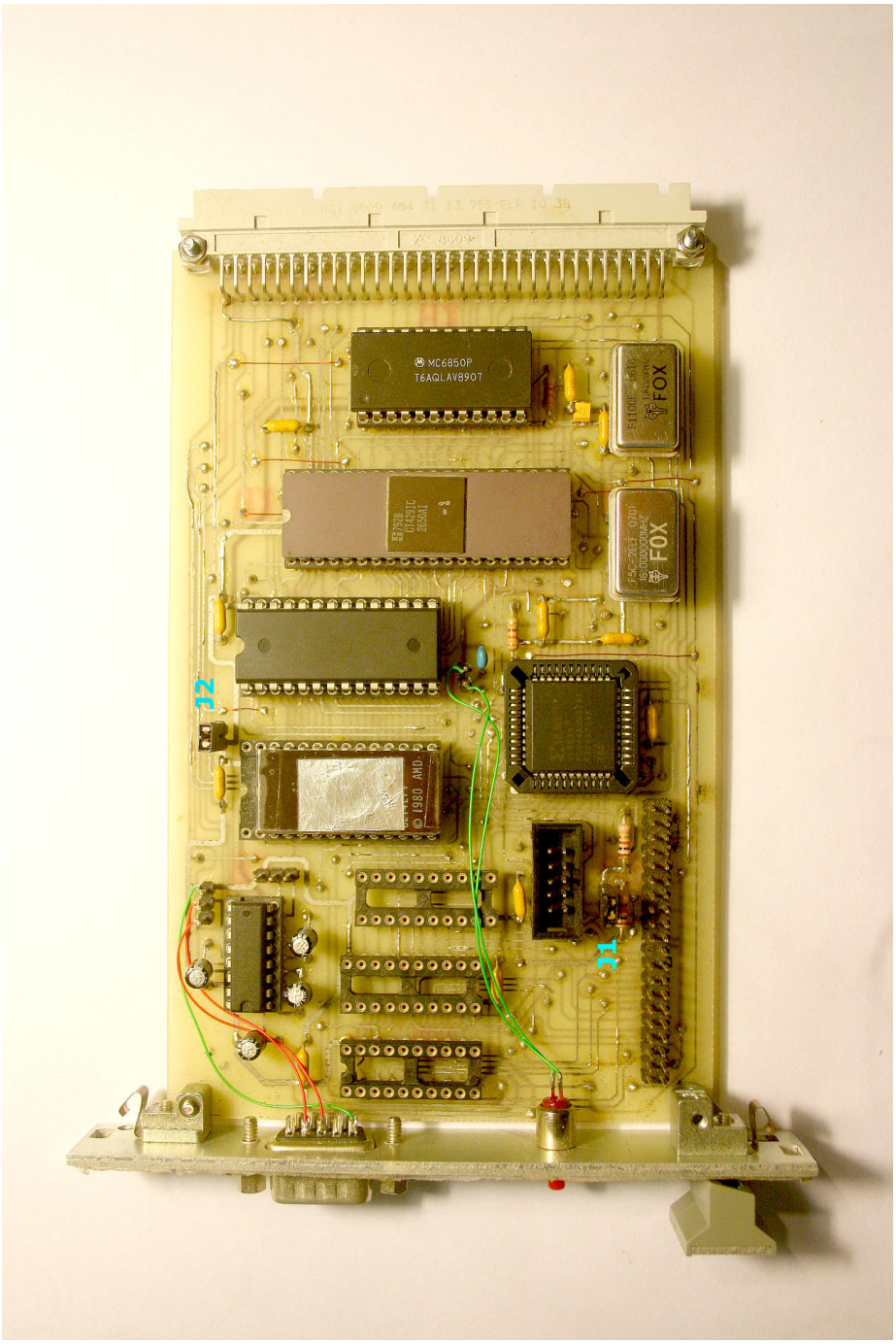
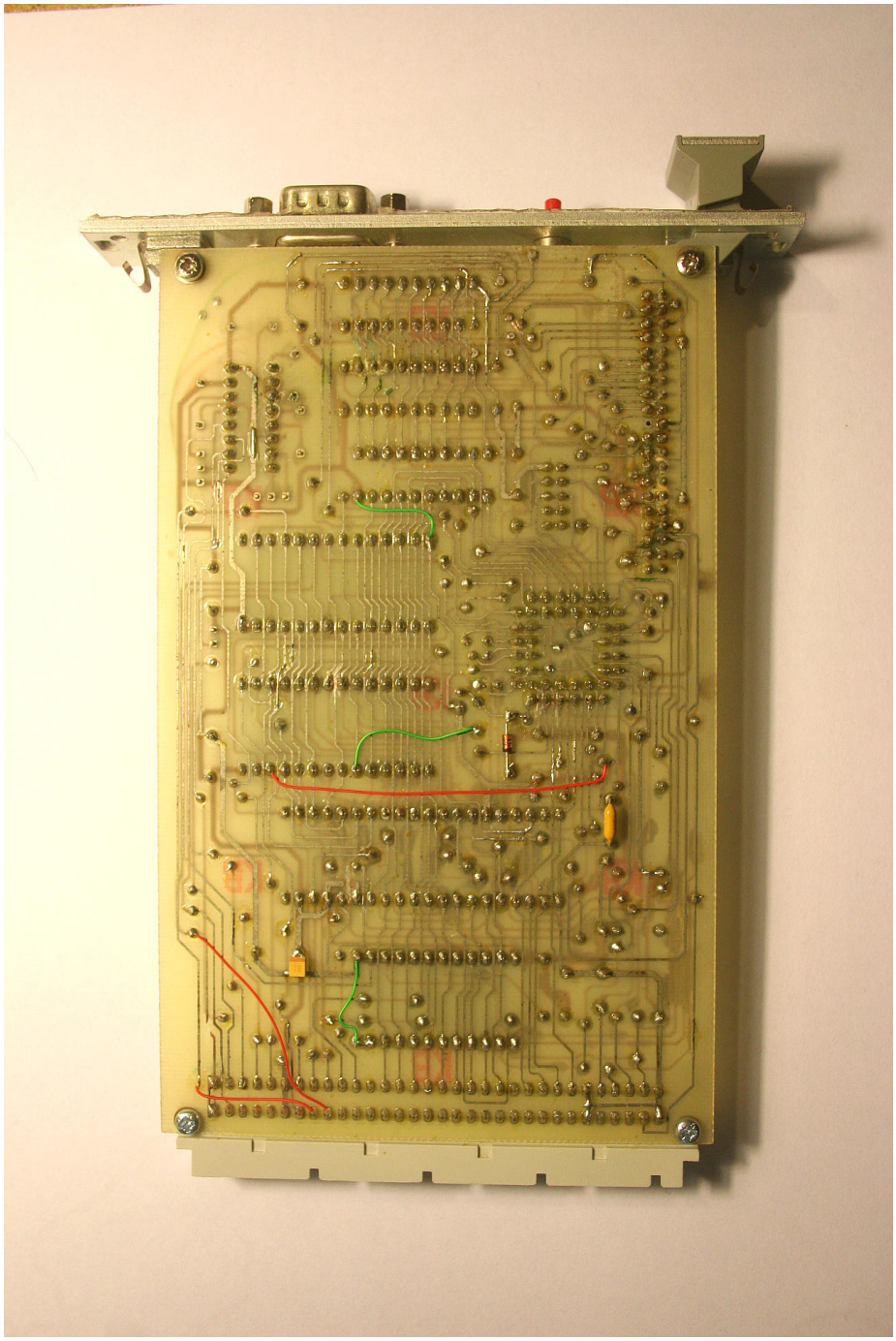


Figure 10: The board solder side, note the corrections done with wire



9 Disclaimer and License

This project been done entierely as an hobby with absolutely NO COMMERCIAL APPLICATION OR INTENT whatsoever , there is absolutely NO INTENT of making any money out of it. This project also been developed during my little free time as a work of passion and love for retrocomputing and old hardware, it's been made at best but don't expect it to be perfect or faults free.

I assume NO responsibility of any sort for damages and / or any improper use of this documentation, feel free to browse it and have fun and interest as much as I do but please accept it as-it-is. My hope is all this can be inspirational to others to continue the study and presevation of interesting technology.

For the sake of clarity I declare this work to be under the Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0) check the Creative Commons website (http://creativecommons.org) if you need details about what this means.