Low Cost Configuration of SRAM Based ALTERA devices

Author: Guillermo Jaquenod, <guillermoj@elkonet.com>
ELKO Componentes Electrónicos S.A.
Constitución 3040 – Ciudad Autónoma de Buenos Aires
C1254AAZ
ARGENTINA

Abstract: *During device operation, SRAM based ALTERA devices store configuration data in volatile SRAM cells, therefore that information must be reloaded each time the device powers up. The configuration data to be loaded must be retrieved from some non-volatile source such as special configuration memories or external sources, using different configuration schemes.*
The use of Configuration devices is very direct and simple, but these benefits are sometimes obscured by its cost, that can get over the cost of the device to be configured. The other configuration methods are a bit more complex and require the use of some I/O pins of an external microprocessor, but they can be useful to attain simple and cheap configuration methods.
This paper describes the use of standard, very low cost serial EEPROM memories for configuration, using minimum microprocessor resources, in Passive Serial Configuration mode; in this case, the EEPROM memory can be used not only to store configuration data but also some other non-volatile information required by the microprocessor.

1. Introduction

During device operation, SRAM based ALTERA devices store configuration data in volatile SRAM cells, therefore that information must be reloaded each time the device powers up. That configuration data must be retrieved from some non-volatile source such as special Configuration Devices or through an external microprocessor, using different configuration schemes (Table 1):

Table 1: Configuration Modes

Configuration scheme	Device Family	Typical use
Configuration device (CD)	APEX20K, FLEX10K ACEX1K, FLEX6000	Configuration with the EP16, EP8, EPC2, EPC1 or EPC1441 configuration devices
Passive Serial (PS)	APEX20K, FLEX10K ACEX1K, FLEX6000	Configuration with a serial synchronous microprocessor interface and the MasterBlaster, ByteBlasterMV or BitBlaster
Passive Parallel Synchronous (PPS)	APEX20K ACEX1K, FLEX10K	Configuration with a parallel synchronous microprocessor interface
Passive Parallel Asynchronous (PPA)	APEX20K ACEX1K, FLEX10K	Configuration with a parallel asynchronous microprocessor interface, that treats the target device as memory
Passive Serial Asynchronous (PSA)	FLEX6000	Configuration with a serial asynchronous microprocessor interface
Joint Test Action Group (JTAG)	APEX20K ACEX1K, FLEX10K	Configuration through the IEEE 1149.1 pins

The use of configuration memories (CD method) such as the EP1441, EPC1, EPC2, EPC8 or EPC16 is very straightforward, but their benefits may be obscured by it cost, which in the case of ACEX1K devices can get over the cost of the device to be configured. As an additional drawback, configuration memories can be only used to store configuration data, and nothing more.

The other configuration methods are a bit more complex and may require the use of a few I/O pins of an external processor, but they can still be useful to attain simple and cheap configuration methods. Specifically, the availability of standard, very low cost EEPROM memories with serial interfaces can offer new alternatives to the configuration problem; this solution requires minimum microprocessor resources, allowing the processor to use the EEPROM memory to store not only configuration data but also some other non-volatile information required by the application.

2. Configuration File Sizes

Each different device requires configuration files of different size, as described in Table 2:

Table 2: Configuration File Sizes

Device	Data Size (Bits)	Data Size (Kbytes)
EP20K400	3,878,000	474
EP20K200	1,950,000	238
EP20K100	***985,000***	***121***
EPF10K250A	3,300,000	403
EPF10K200E	2,757,000	337
EPF10K130E	1,840,000	225
EPF10K130V	1,600,000	194
EPF10K100E	1,336,000	164
EPF10K100, EPF10K100A, EPF10K100B	1,200,000	146
EPF10K70	***892,000***	***109***
EPF10K50E, EP1K50	***785,000***	***96***
EPF10K50, EPF10K50V	***621,000***	***76***
EPF10K40	***498,000***	***61***
EPF10K30E, EP1K30	***470,000***	***58***
EPF10K30A	***406,000***	***50***
EPF10K30	***376,000***	***46***
EPF10K20	***231,000***	***29***
EPF10K10A, EP1K10	***120,000***	***15***
EPF10K10	***118,000***	***15***
EPF6024A	***398,000***	***49***
EPF6016, EPF6016A	***260,000***	***32***
EPF6010A	***260,000***	***32***

These numbers can be used only as an estimate because different MAX+plus II or Quartus II software versions may add a slightly different number of padding bits during programming. However, it must be noted that most of the cost-sensitive devices (typed in italics) have configuration file sizes under one Mbit.

3. The PASSIVE SERIAL configuration mode

In PASSIVE SERIAL (PS) configuration mode, an external controller passes configuration data to one or more devices via a serial data stream. The hardware resources required by the

controller to configure such devices are very reduced, as shown in Figure 1, involving only the use of five lines: two bi-directional lines (CONF_DONE and nSTATUS), and three device input lines (DATA, nCONFIG and DCLK).

PS configuration mode can be used to configure merely one device, or a chain of devices; in the last case, the nCEO output pin form the first device is cascaded into the nCE input pin of the second device, and so on. It is a three steps process:

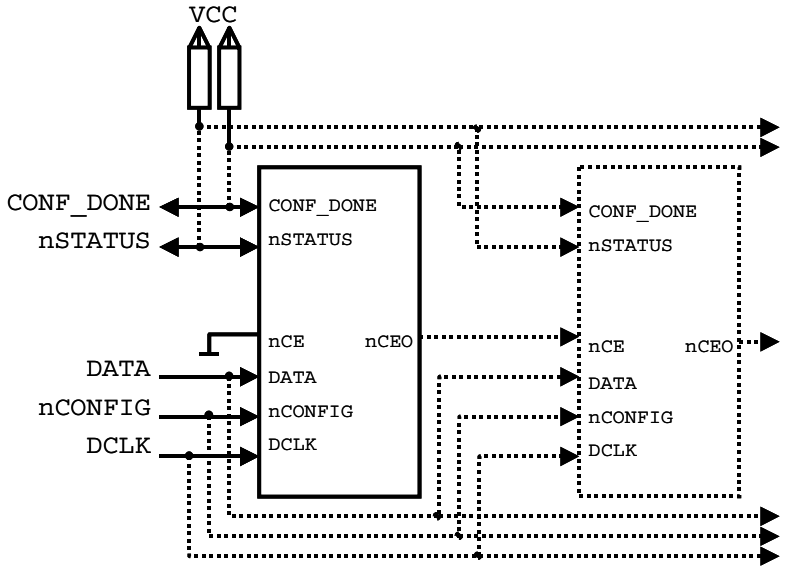

FIGURE 1

- Start: the controller checks that nSTATUS is high, puts a zero on nCONFIG, waits until CONF_DONE and nSTATUS go to zero, then writes a one on nCONFIG and waits until the target device releases nSTATUS.

- Data transfer: after the configuration process has started, the microprocessor places the configuration data one bit at a time, generating DCLK positive edges to input this data in the target devices, with a clock frequency up to 16MHz. Following the falling edge of DCLK after all it configuration bits are received, the device just configured releases CONF_DONE and pulls down it nCEO output. While loading data, the controller must check the nSTATUS line, which is pulled down by the device under configuration if it detects some error. This error detection capability is obtained through checksum information embedded with the configuration data.

- Initialization: after CONF_DONE goes high, DCLK must be clocked 40 additional times for APEX devices and 10 times for FLEX devices to initialize the device, going into user mode. When multiple devices are configured, the wired-AND CONF_DONE line remains low until the last device is configured, therefore all devices go to Initialization mode at the same time.

A timing diagram of the configuration process is shown in Figure 2, highlighting the different steps. From a micro controller point of view (where time units are usually measured in microseconds) the time constraints to be satisfied are:

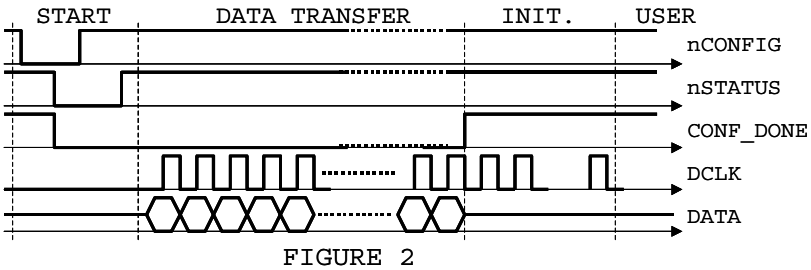

FIGURE 2

$\mathbf{t_{CFG}}$: nCONFIG low pulse width must be greater than 8 microseconds

$\mathbf{t_{STATUS}}$: the device can take up to 40 microseconds to release nSTATUS

$\mathbf{t_{CF2CK}}$: the delay from nCONFIG high until the first rising edge on DCLK must be greater than 40 microseconds

There is a point to be noted about the PS mode: after configuration has ended, any change in the DCLK or DATA lines is ignored by the target devices. This point can be used to share these microprocessor lines with other applications.

4. About .HEX, .TTF and .RBF files

After project compilation, the MAX+PLUS II Compiler automatically generates SRAM Object Files (.SOF), Tabular Text Files (ASCII file with extension .TTF), and Hexadecimal Files (ASCII file with extension .HEX), that contain the data for device configuration in a PS configuration scheme. By using the Convert SRAM Object Files command (File menu), the designer can also create a "raw binary file" (with extension .RBF), that stores the configuration data in binary format.

In all cases, the configuration data is represented as bytes that must be sent to the configured device with the LSB first.

HEX: it is an ASCII text file (with the extension .hex) in the Intel Hex format, automatically generated by the Assembler when an SRAM device is assigned as the target device. Many SRAM Object Files (.SOF) generated during compilation may be combined in only one HEX file with the Convert SRAM Object Files command.
An extract of a sample HEX file is shown:

```
:020000040000FA
:2B000000FFFF62FF2500FFFFFFFFFFFFFFFFFFFF...FF76
:2B002B006018866118866118866118866118866188...DDF9
...
:2B397200601886611886611886611886611886611888...A523
:02399D00FFFF2A
:00000001FF
```

If the second line is analyzed:

```
:     Signals a line start
2B    Number of data bytes in this line (hex 2B=43 decimal)
0000  Address of the first data byte
00    Used to identify programming data
FF    1st data byte (255 decimal)
FF    2nd data byte (255 decimal)
62    3rd data byte ( 98 decimal)
FF    4th data byte (255 decimal)
25    5th data byte ( 37 decimal)
00    6th data byte (  0 decimal)
FF    7th data byte (255 decimal)
...   intermediate bytes
FF    43th (last) data byte
CC    Line Checksum
```

In the same way, the last programming data line `(:02399D00FFFF2A)` has only two bytes with value FF, FF.

TTF: it is an ASCII text file (with extension .TTF) that stores the configuration data in a tabular format. If the same design is analyzed, this text file starts as follows:

```
255,255, 98,255, 37,  0,255,255,255,255,255,255,255,255,...
```

In the TTF file, each configuration data byte is written using "packs" of three characters in decimal format (left padded with spaces), with a comma as separator between packs.

RBF: the RBF is a binary file, where each configuration data byte is stored using only one file byte. The RBF file of this design, inspected with a binary editor, could be read as:

```
0000 FF FF 62 FF 25 00 FF FF FF FF ...
```

A RBF file is the binary equivalent of a TTF file.

To write the EEPROM, the information stored in these files must be bit reversed before programming, because when the EEPROM is read the first output data bit is the MSB, whereas ALTERA devices require the LSB to be the first.

5. The operation of the I^2C EEPROM memories

Serial EEPROM memories fall in two main categories: I^2C (2 wires interface) and SPI or MICROWIRE (3 wires interface). In the case of I^2C memories this interface between a controller and the EEPROM memory is achieved through the use of the lines SCL and SDA.

The I^2C standard defines four types of "agent" on the bus:

- **Transmitter:** The device which is currently sending data to the bus
- **Receiver:** The device which is currently receiving data from the bus
- **Master:** The device which initiates a transfer, generates clock signals and terminates a transfer.
- **Slave:** The device addressed by the MASTER.

During I^2C communications, data on the SDA pin may change only during SCL low time periods, being sampled in the rising edge of SCL; however, the exceptional change of SDA during SCL high periods is used to indicate a START or STOP condition. All I^2C transactions must begin with a START condition and terminate with a STOP condition; they are generated by the master and are defined as:

- A START condition is signaled by a High-to-Low transition on SDA with SCL high; it must precede any other command.
- A STOP condition is signaled by a Low-to-High transition on SDA with SCL high.

Address and Data words are serially transmitted to and from the EEPROM in 8-bit words, fo;;owed by a ninth clock pulse on SCL (generated by the MASTER), when the transmitter must release the SDA line allowing the receiver to pull it down to ACKnowledge that it has received the word. After the transmitter receives the ACK (usually sent after each byte of data) the transmitter can continue sending data. To avoid contentions, SDA operates as an Open Drain output, making mandatory the use of a Pull-Up resistor.

For every transaction one control byte is sent after a START condition. This control byte consists of a unique 7-bit device address and a R/W (read/write) bit.

Some devices have address pins which allow you to change the address of a device. For example, an EEPROM with three address pins A2, A1, A0 pins connected to GND would have a address of 1010*000* while if A2,A1,A0 are connected to Vcc the address would be 1010*111*. Other devices use these bits in the control byte as "page selection bits".

In the case of the AT24C1024 memory, a 1024Kbit EEPROM (organized as 128 Kbytes), the control byte has the format shown in Figure 3.

1	0	1	0	0	A_1	P_0	R/W

FIGURE 3

- This 1024K EEPROM uses the one device address bit A_1, allowing the connection of two devices on the same bus, and this bit must compare to the corresponding hardwired input pin.
- The seventh bit P_0 is the memory page address bit, being the 17th address bit of the data word address that follows.
- The eighth bit R/W selects a read operation (if R/W=1) otherwise a WRITE.

After the control byte is sent, the receiver sends an acknowledgement ("ACK") to the transmitter. Figure 4 shows the waveforms related to the control byte transmission process:

- In the starting point SCL is low, and SDA is in tri-state. The pull-up resistor drives SDA to one.
- The processor drives SCL high.
- A START is initiated when the processor drives SDA low while SCL is high.
- The processor drives SCL low.
- To write a one, the processor drives SDA to tri-state, because the pull-up will drive SDA high.

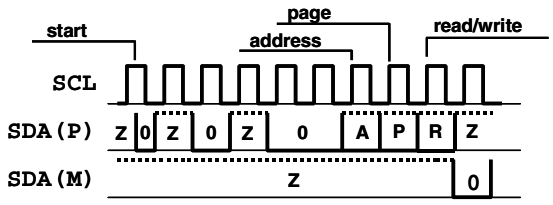

FIGURE 4

- A complete positive SCL pulse is issued
- The processor drives SDA low, and a complete positive SCL pulse is issued
- The processor SDA goes to tri-state, the pull-up drives SDA high, and a complete positive SCL pulse is issued.
- The processor drives SDA low, and two complete positive SCL pulse are issued
- The processor drives SDA to tri-state or to zero, according the A1 value, and a complete positive SCL pulse is issued
- The processor drives SDA to tri-state or to zero, according the Page Select (P0) value, and a complete positive SCL pulse is issued
- The processor drives SDA to tri-state or to zero, according the R/W value, and a complete positive SCL pulse is issued
- The processor drives SDA to tri-state. Simultaneously, the SDA line is pulled low by the EEPROM to indicate an acknowledge.
- When driving SCL high, the processor captures the SDA value. It must be zero, otherwise it signals an error.
- When SCL is driven low, the EEPROM releases SDA to tri-state and the high memory address byte transfer may start.

There are two write methods and three read methods to access the EEPROM contents:

- **BYTE WRITE:** a ***byte write*** operation is used to put data into a specific memory location. A ***byte write*** operation requires a START, the 8-bit device address (with R/W=0 and P0 set to the value of the 17th address bit), the ACK from the receiver, the high byte of the address, the ACK from the receiver, the low byte of the address, the ACK from the receiver, the data to be written, the ACK from the receiver, and a STOP. Once the STOP is received, an internal write process is triggered and the EEPROM will not respond until the write is complete.
- **PAGE WRITE:** a ***page write*** operation is be used to program up to 256 bytes at a time in a given page. It is initiated in the same way as a byte write, but the controller does not send a STOP condition after the first data word is clocked in. Instead, after the EEPROM acknowledges receipt of this data word, the controller can transmit up to 255 more data

words (waiting for the ACK after each byte is sent). To terminate the ***page write*** sequence the controller issues a STOP condition. The data word address lower 8 bits are internally incremented following the receipt of each data word, but the higher data word address bits remain unmodified; thus, when the low data word address reaches 255, the following byte is placed at the address 0 in the same page.

- **CURRENT ADDRESS READ:** the internal data word address counter maintains the last address accessed during the last read or write operation, incremented by one; this address stays valid between operations as long as the chip power is maintained. For a ***current address read*** the controller sends a device address with the R/W select bit set to '1' receives the acknowledge from the EEPROM, but then it is the EEPROM who takes the transmitter role sending the requested data to the controller (note that the controller does not send any address). To close the ***current address read*** operation, after the EEPROM sent a byte the receiver (the controller in this case) does not respond with an ACK but issues a STOP condition. This is called a "NACK".

- **RANDOM READ:** a random read requires a ***dummy byte write*** sequence to load in the internal data word address register followed by a ***current address read***. Following a ***byte write*** command and after the device address word (with R/W=0) and data word address have been clocked in and acknowledged by the EEPROM, the controller aborts the ***byte write*** generating another START condition. The controller now initiates a ***current address read*** by sending a device address (now with R/W=1). The EEPROM acknowledges the device address and serially clocks out the data word. The controller sends a NACK and the transaction ends.

- **SEQUENTIAL READ:** ***sequential reads*** are initiated by either a current address read or a random address read. After the controller receives a data word, it respond with an acknowledge. As long as the EEPROM receives an acknowledge it will continue incrementing the data word address and serially clock out sequential data words. When the whole memory address limit is reached the data word address will rollover and the sequential read will continue. The sequential read operation is terminated when the micro controller respond with a NACK.

A micro controller routine may be easily written based on four procedures (Figure 5):

- **WRITE_ZERO:** (waveform A) drives the SDA line to '0', drives SCL to '1', then to '0', then SDA to tri-state, and returns no value
- **WRITE_ONE:** (waveform B) puts the SDA processor output in tri-state, captures the value of the SDA line, drives SCL to '1', drives SCL to '0', and returns the captured SDA value.
- **START:** (waveform C) drives SDA to tri-state and SCL='1', then SDA to '0', then SCL to '0', then SDA to tri-state, and returns no value
- **STOP:** (waveform D) drives SDA to '0', then SCL to '1', then SDA to tri-state, and returns no value

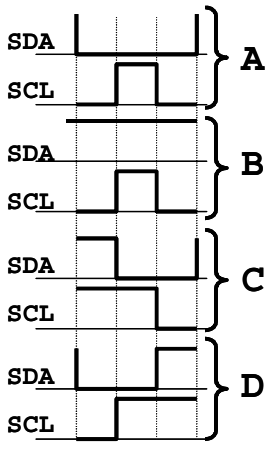

FIGURE 5

In this case every transaction starts and end with SCL and SDA in '1'; within a transaction, each bit period starts and ends with SCL='0' and SDA=tri-state.

Today, the size of higher density EEPROM serial memories with I^2C interface is around 1 Mbit (e.g. Atmel AT24C1024); a device of this size can be used to store the configuration file for any FLEX6000 devices, FLEX10K devices ranging from 10K10 to 10K70, ACEX1K devices from

EP1K10 to EP1K50, and APEX devices from EP20K30E to EP20K100/100E; lower size memories (e.g. the Microchip 24LC128, with 128 Kbits), can be used for the ALTERA smaller devices such as EP1K10 or FLEX10K10.

However, I^2C memories are not the only alternative. If other interfaces are considered, many other choices exist. As an example, ST Microelectronics has announced serial FLASH memories with up to 8 Mbits; these devices, controlled through an SPI interface, can be also easily interfaced to a CPLD.

6. The proposed circuit

Figure 6 shows an hypothetical circuit using an I^2C serial EEPROM, showing that only one pull-up has been added to the standard configuration circuit required for CPLDs. The only point to note is that the EEPROM and the CPLD use different clocks (DCLK and SCL), because during the ACK reception and EEPROM address settings, SCL must pulse and DCLK must remain quiet.

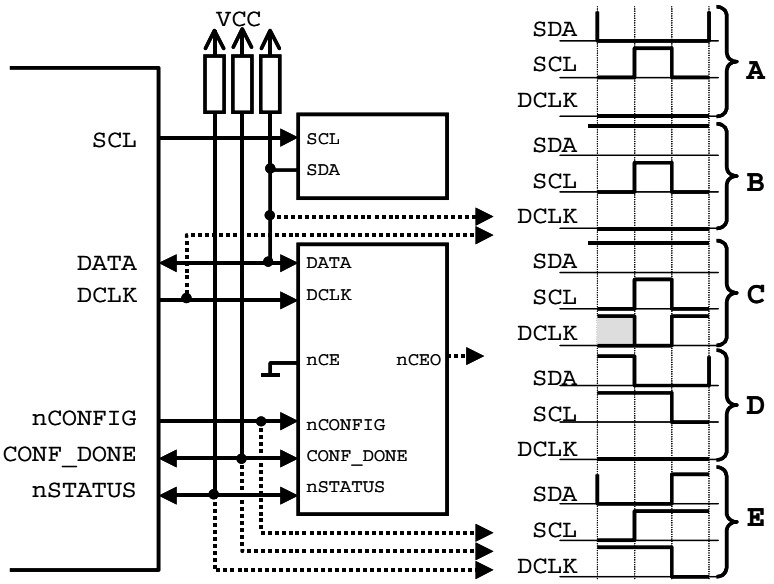

FIGURE 6

Figure 6 shows also the five procedures now required to manage the configuration process:

- **A. WR_0:** the EEPROM is written with a zero, but DCLK remains unchanged
- **B. WR_1/READ_ACK:** used to write a one on the EEPROM or to read the ACK from the EEPROM, DCLK remains unchanged, on the rising edge captures and then returns the SDA value
- **C. RD:** used to read one bit from the EEPROM and to input this bit on the CPLD, returns the nSTATUS value
- **D. START:** issues a START condition on the EEPROM, and DCLK remains in zero.
- **E. STOP:** issues a STOP condition on the EEPROM, returning DCLK to zero

In the case of SPI/Microwire memories, Data Input (DI) and Data output (DO) use different lines, therefore the interface requires one additional micro controller pin. However, configuration may be faster since no ACK pulses are generated by the EEPROM.

7. Pseudo code of the configuration procedure

The configuration process may be described using the previous five procedures, as follows. The time delays are for the worst case and may be smaller, since they run concurrently with other procedures.

```
------------------------------------------------------------------------
-- the CPLD configuration process is triggered

nCONFIG = 0; wait until nSTATUS=0; wait for 8 us;
nCONFIG = 1; wait for 40 us until nSTATUS=1;
```

```
if (nSTATUS=0 OR CONF_DONE=1) abort();
wait for 40 us;

-------------------------------------------------------------------------
-- EEPROM address setting. It starts with a dummy byte write

start();
wr_1(); wr_0();wr_1(); repeat 5 wr_0(); -- control byte B"10100000"
if wr_1() /= 0 abort(); -- check ack
repeat 8 wr_0(); -- high address byte
if wr_1() /= 0 abort(); -- check ack
repeat 8 wr_0(); -- low address byte
if wr_1() /= 0 abort(); -- check ack

-- byte write is aborted with a new START, and a sequential read starts

start();
wr_1(); wr_0();wr_1(); repeat 4 wr_0(); wr_1();-- control byte B"10100001"
if wr_1() /= 0 abort(); -- check ack

-------------------------------------------------------------------------
-- data read from the EEPROM is clocked in the CPLD, one byte at a time
-- the process continue until the CPLD puts CONF_DONE high
-- this WHILE loop is repeated tens to hundreds thousand times!!!
-- it's the critical part to be optimized to reduce configuration time

while CONF_DONE=0 loop begin
  repeat 8 if rd()=0 abort(); -- check nSTATUS changes
  if wr_1() /= 0 abort(); -- check ack
  end;

-------------------------------------------------------------------------
-- 40 additional CPLD clocks are issued to complete configuration

repeat 40 if rd()=0 abort(); -- check nSTATUS changes
stop();
```

8. Conclusions

Configuration of SRAM based ALTERA devices can be easily done by a microcontroller using the PS mode, and sharing many lines with other serial devices. In fact, only three lines (nCONFIG, CONF_DONE and nSTATUS) are exclusively devoted to the configuration process. The availability of low cost micro controllers makes this configuration method a valid alternative to configuration memories.

9. Acknowledgments

The idea and first application of this configuration method were developed by Julian Kobak and Marcelo Navarro, from Videotron S.A. (ARGENTINA).

10. References

- Altera AN116."Configuring APEX20K, FLEX10K & FLEX6000 Devices", June 1999, v1.0.
- Altera DS-EPROM-09. "Configuration EPROMs for FLEX Devices", October 1998, ver.9.
- Atmel Data Sheet 1471D-07/01. "AT24C1024 Data Sheet".
- Philips Specification. "The I^2C-bus and how to use it". December 1998