sockit_owm
1-wire (onewire) master

Copyright (CC BY-SA 3.0) 2010-2011

Iztok Jeras

Project home pages:

https://github.com/jeras/sockit_owm

http://opencores.org/project,sockit_owm

Table of Contents

1 Introduction....5
  1.1 1-wire protocol and devices....5
  1.2 Features....5
  1.3 List of source files....5
2 Verilog module parameters and ports....6
  2.1 Parameters....6
    2.1.1 Optional functionality....6
    2.1.2 Data bus width, number of wires and overdrive enable....6
    2.1.3 Clock divider ratio and base time periods....6
  2.2 Ports....7
    2.2.1 CPU bus interface....8
    2.2.2 1-wire interface....8
  2.3 Metastability evaluation....9
3 Supported 1-wire functionality....10
  3.1 RTL description....10
  3.2 1-wire cycle timings....10
4 Processor interface....13
  4.1 Address space....13
  4.2 Clock dividers....13
  4.3 1-wire line multiplexing and power enable....14
  4.4 Control/status register....14
  4.5 Driver access sequences....15
    4.5.1 Polling routines....16
      Reset and presence pulses....16
      Read/write data slots....16
      Delay of 0ms or 1ms....17
    4.5.2 Interrupt routines....17
      Reset and presence pulses, read/write data slots....17
      OS initialization and interrupt service routine....18
5 Software driver....19
  5.1 Port of Public Domain Kit components to Nios II HAL and µC/OS-II....19
    5.1.1 Nios II HAL and µC/OS-II drivers....19
    5.1.2 Public Domain Kit components....19
  5.2 Adding support for new devices....20
  5.3 Possible improvements....20
6 Altera development tools integration....21
  6.1 SOPC Builder and Nios II EDS integration....21
    6.1.1 SOPC Builder....21
      Adding the component to a project....21
      Configuring the component....21
    6.1.2 Nios II EDS....22
7 Demo hardware and software implementation....23
  7.1 Demo hardware....23
    7.1.1 Demo hardware logic consumption....23
  7.2 Demo software....24
    7.2.1 Demo software memory consumption....24

8 Testing....25
8.1 Verilog RTL simulation....25
8.1.1 1-wire slave model....25
8.1.2 Verilog testbench....26
8.1.3 Parameter tests....26
8.2 C driver tests....27
8.3 Testing TODO....27
9 License....28
10 References....29

Index of Tables

Table 1: File list....5
Table 2: List of sockit_owm module parameters....6
Table 3: Base time period and base frequency options....7
Table 4: List of sockit_owm module ports....7
Table 5: Wishbone equivalents of Avalon MM signals....8
Table 6: 1-wire protocol standard timing restrictions....11
Table 7: Timing options....11
Table 8: 32bit interface address space....13
Table 9: 8bit interface address space....13
Table 10: Control/status register structure....14
Table 11: Control/status register bits with descriptions....14
Table 12: List of supported cycles....15
Table 13: Files for Nios II HAL and µC/OS-II integration....19
Table 14: Files from the Public Domain Kit....20
Table 15: Driver configuration options....22
Table 16: Pin-out of the PS/2 connector on the Terasic DE1 board....23
Table 17: Demo software main function files....24
Table 18: Verilog RTL testbench files....25

Drawing Index

Drawing 1: Bus timing (read/write cycle and interrupt status)....8
Drawing 2: 1-wire line driver schematic....9
Drawing 3: RTL block diagram....10
Drawing 4: Reset and presence timing diagram....12
Drawing 5: Write 0 timing diagram....12
Drawing 6: Write 1, read timing diagram....12
Drawing 7: Write 1 or read timing diagram....12
Drawing 8: Component GUI for sockit_owm....22
Drawing 9: Image of the PS/2 connector on the Terasic DE1 board....23
Drawing 10: 1-wire slave model timing diagram....25

Abbreviations, terminology and product names

Verilog HDL	Verilog hardware description language
RTL	register transfer level
SoC	system on chip
CPLD	complex programmable logic device
FPGA	field-programmable gate array
ASIC	application-specific integrated circuit
BSP	board support package
HAL	hardware abstraction layer
OS	operating system
µC/OS-II	preemptive, real-time deterministic multitasking kernel written in ANSI C
1-wire (onewire)	single wire serial bus protocol
Quartus II	FPGA design and synthesis tool from Altera
SOPC Builder	system on a programmable chip builder
Nios II EDS	Nios II (32-bit soft-core processor from Altera) embedded development suite
Avalon MM	Avalon memory mapped switch fabric interface
Wishbone	open source hardware computer bus
PS/2	PS/2 connector (6-pin Mini-DIN)

1 Introduction

This document describes a 1-wire master written in Verilog HDL, ready for integration into a FPGA or ASIC based SoC. A port of the 1-wire Public Domain Kit (version 3.10r2) from Maxim is also provided, with all the code required for integration into the Altera development environment.

1.1 1-wire protocol and devices

The 1-wire protocol was defined by Dallas Semiconductor. In 2001 the company was bought by Maxim Integrated Products, Inc. Maxim is the main manufacturer of 1-wire and iButton devices. Devices and standards can be found on their web-page http://www.maxim-ic.com/products/1-wire/.

1.2 Features

Only the physical layer of the protocol is implemented in hardware, so only bit level data transfers are possible, a software driver or hardware state machine is required to perform byte level transfers and to support higher protocol layers. Features:

1. reset pulse generation and presence detection, bit level write and read, timed idle cycle
2. overdrive mode
3. from 1 to 16 wires (4 wires for an 8bit interface)
4. power (strong pull-up)
5. polling or interrupt driven CPU interface

There are no special features, like active pull-up or adaptive timing, for handling a long and unstable bus. Hardware interrupt (rarely used part of a reset cycle) is also not supported.

1.3 List of source files

The sources can be divided into Verilog RTL and testbench code, C code for drivers, code required for integration into Altera development tools and documentation.

Table 1: File list

file name	description
`doc/onewire.odt`	this document (onewire.odg contains source drawings)
`hdl/onewire_tb.v`	Verilog testbench
`hdl/onewire_slave_model.v`	Verilog model of 1-wire slave device (very simplified)
`hdl/sockit_owm.v`	main module Verilog RTL source
`hdl/wishbone2bus.v`	Wishbone to bus protocol converter Verilog RTL source
`sockit_owm_hw.tcl`	Script for integrating the hardware into Altera dev. tools
`sockit_owm_sw.tcl`	Script for integrating the software into Altera dev. tools
`inc/sockit_owm_regs.h`	C header file describing CPU bus accessible registers
`HAL/inc/sockit_owm.h`	C header file for Nios II HAL integration
`HAL/src/owlnk.c, ...`	C source code ported from the 1-wire Public Domain Kit

2 Verilog module parameters and ports

The next section describes how to properly integrate the Verilog module into a SoC.

2.1 Parameters

The next list of parameters is almost complete, but there are some timing parameters that should be modified only by developers able to read the Verilog HDL source code.

Table 2: List of sockit_owm module parameters

parameter	type	default	range (8bit interface)	description
`OVD_E`	integer	1	0, 1	overdrive functionality enable
`CDR_E`	integer	1	0, 1	clock divider register enable
`BDW`	integer	32	32 (8)	bus data width
`OWN`	integer	1	1, 2, ..., 16 (4)	number of 1-wire lines
`BTP_N`	string	"5.0"	"7.5", "5.0", "6.0"	base time period for normal mode
`BTP_O`	string	"1.0"	"1.0", "0.5"	base time period for overdrive mode
`CDR_N`	integer	15	0, 1, ..., 65535 (255)	clock divider ratio for normal mode
`CDR_N`	integer	2	0, 1, ..., 65535 (255)	clock divider ratio for overdrive mode

2.1.1 Optional functionality

If overdrive mode is not required it can be disabled (`OVD_E=0`).

If the input clock is constant, clock divider ratios can be implemented as constant parameters (`CDR_E=0`) instead of registers (`DCR_E=1`).

2.1.2 Data bus width, number of wires and overdrive enable

The width of the data bus is 32bit by default. An 8bit bus option is provided for connecting to to a 8bit CPU, but this restricts the number of 1-wire lines to 4 (`OWN<=4`). The details are explained in the section on module ports.

For 32 bit CPU interfaces if is possible to have from 1 to 16 1-wire lines. Note that only one line can be active at the same time. If simultaneous data transfers on more than one line are required, one solution would be to instance more than one `sockit_owm` module, but this would require a software driver modification.

2.1.3 Clock divider ratio and base time periods

The internal state machine requires a base time period. For normal mode the base can be exactly 7.5µs or 5.0µs, or in the range from 6.0µs to 7.5µs. For overdrive mode the base can be 1.0µs or in the range from 0.5µs to 0.66µs. The system clock frequency must be such that it is possible to divide it into the desired base period. If overdrive is enabled, both normal and overdrive mode periods must be achievable. Exact time periods offer a better optimized implementation, range periods should be used only if a round frequency is not possible.

Table 3: Base time period and base frequency options

overdrive support (`OVD_E`)	disabled			enabled		
base time period [µs] (normal mode)	7.5	5	6.0 - 7.5	7.5	5.0	6.0 - 7.5
base time period [µs] (overdrive mode)				1.0	1	0.5 - 0.66
base frequency [Mhz]	0.133	0.2	0.166 - 0.133	2	1	2 - 1.5

The clock divider ratio parameters (register values) are computed with the next formulas (replace the base period as required):

$$\text{CDR_N} = f_{clk} \cdot 5.0\,\mu s - 1$$

$$\text{CDR_O} = f_{clk} \cdot 1.0\,\mu s - 1$$

And the exact base time period is computed as:

$$\text{BTP_N} = \frac{\text{CDR_N} + 1}{f_{clk}}$$

$$\text{BTP_N} = \frac{\text{CDR_N} + 1}{f_{clk}}$$

If the dividing factor is not a round integer, than the timing of the controller will be slightly off, and would support only a subset of 1-wire devices with timing closer to the typical 30µs base. If the relative error is large (this is possible if the system clock frequency is low) than range periods should be used. For range periods longer times (closer to the higher limit) offer a better signal quality, although a lower data rate.

2.2 Ports

Module ports are divided into two groups: CPU interface ports and 1-wire interface ports.

Table 4: List of sockit_owm module ports

signal	width	direction	description
`bus_wen`	`1`	input	bus write enable
`bus_ren`	`1`	input	bus read enable
`bus_adr`	`BAW*`	input	bus address
`bus_wdt`	`BDW`	input	bus write data
`bus_rdt`	`BDW`	output	bus read data
`bus_irq`	`1`	output	bus interrupt request status
`owr_i`	`OWN`	input	1-wire bus status
`owr_e`	`OWN`	output	1-wire open collector driver enable
`owr_p`	`OWN`	output	1-wire power enable (strong pull-up)

* for a 32 bit interface BAW=1, for an 8bit interface BAW=2

2.2.1 CPU bus interface

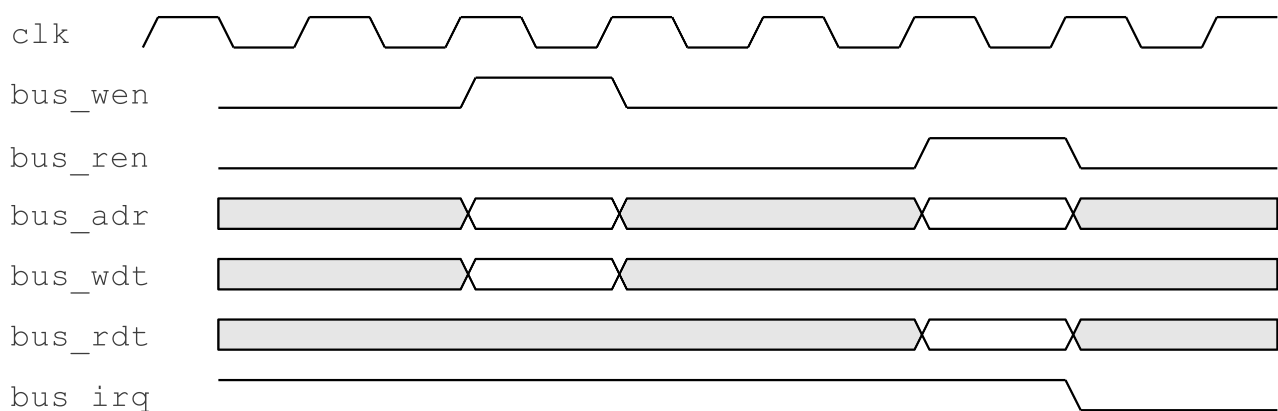
The default bus interface is 32bit Avalon MM compatible. It is trivial to connect it to Wishbone (`wishbone2bus` module) and it should be easy to connect to AMBA APB. The module can be also connected to an 8bit bus (a micro-controller for example).

Table 5: Wishbone equivalents of Avalon MM signals

bus signal	Avalon MM signal	Wishbone signal
bus_wen	avalon_write	cyc & stb & we
bus_ren	avalon_read	cyc & stb & ~we
bus_adr	avalon_address	adr
bus_wdt	avalon_writedata	dat_w
bus_rdt	avalon_readdata	dat_r
bus_irq	avalon_interrupt	irq

All CPU related signals are active high. Both read and write cycles are a single clock period long, the omitted Avalon MM signal `bus_waitrequest` would be a constant '0'. Read data is valid only the first cycle of `bus_ren` being active, it might change in the next cycle due to the interrupt status being cleared.

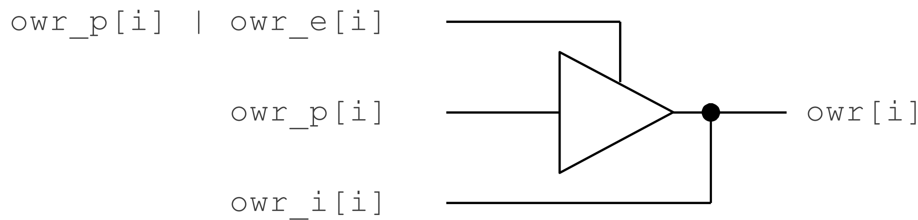
The interrupt signal indicates, that a reset (with presence detection) or a read/write bit 1-wire cycle has finished. After reading the control/status register the interrupt is cleared.



Drawing 1: Bus timing (read/write cycle and interrupt status)

2.2.2 1-wire interface

The 1-wire module requires an external open drain driver to be able to drive a bidirectional pin. The `owr_p` signal can be used to supply extra power to the 1-wire line (for temperature conversion, ...), if the signal is not used, 0 should be applied instead. The next scheme shows the driver for the i-th line.



Drawing 2: 1-wire line driver schematic

The next Verilog code should be used for the i-th line on the top level of the SoC.

```
assign owr  [i] = (owr_p[i] | owr_e[i]) ? owr_p[i] : 1'bz;
assign owr_i[i] =  owr;
```

And if the power signal is ignored.

```
assign owr  [i] = owr_e[i] ? 1'b0 : 1'bz;
assign owr_i[i] = owr;
```

On some FPGA it is possible to set driver strength at compile time. There are no extra features for long 1-wire networks, but it is suggested to use weak drivers (slow slew rate) to minimize transitional effects on the network. If the FPGA pin is to be used to provide power to an 1-wire slave, the opposite, to use strong drivers is suggested. If both requirements are present the user must choose the proper compromise based on specifications and testing.

2.3 Metastability evaluation

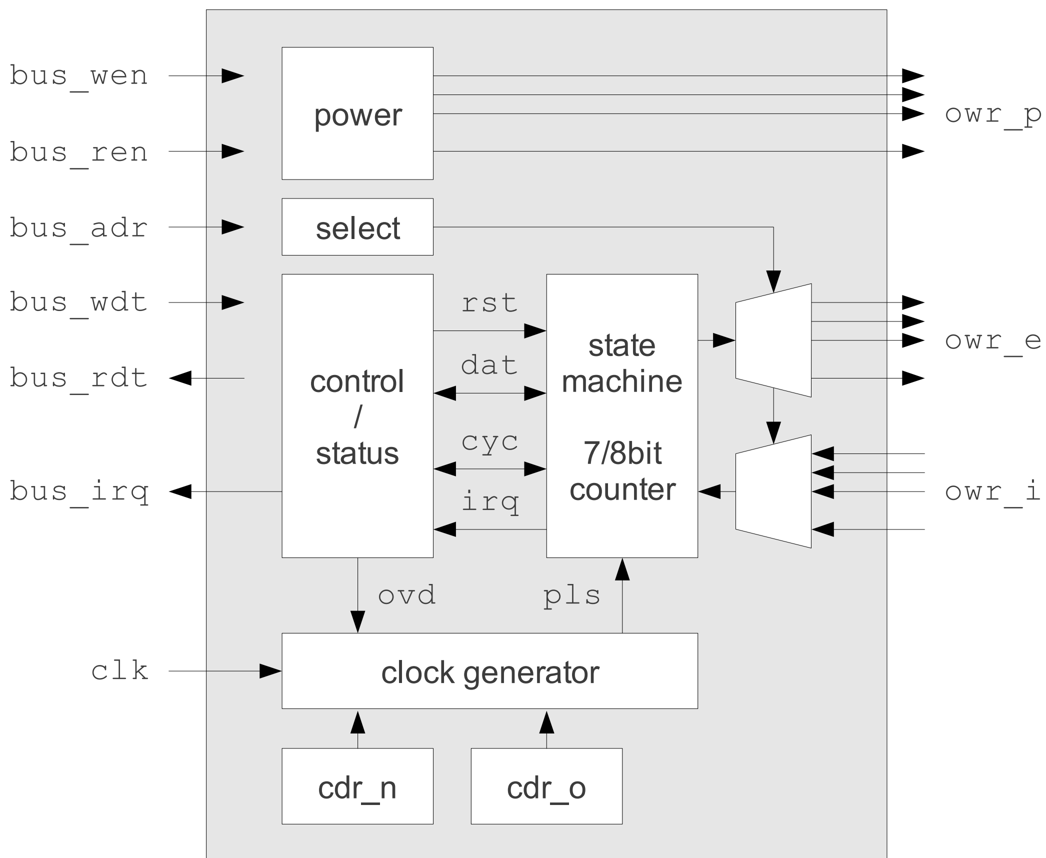
The 1-wire line is sampled at the presence detection point for reset cycles and at the data sample point for read/write data cycles. The sampled value is first stored in a register and later at the end of the 1-wire cycle copied into a register visible on CPU interface data bus. The shortest time between sampling the 1-wire line and data being available on the CPU bus is 5µs, this is enough for all metastability problems to level out.

3 Supported 1-wire functionality

The `sockit_owm` module provides timed cycles for 1-wire reset/presence and write/read bit for both normal and overdrive mode. This section describes how the 1-wire link layer is implemented by the `sockit_owm` module RTL.

3.1 RTL description

The core of the RTL is a state machine with a 7bit or 8bit counter. On each bus write the counter and `owr` signals are set to appropriate values. The clock generator provides pulses every base time period (depends on overdrive mode being used), on each pulse the counter is decremented and the `owr` signals are changed as required. More details can be found in the RTL source.



Drawing 3: RTL block diagram

3.2 1-wire cycle timings

The timing for 1-wire cycles is defined as a set of minimum and maximum values for certain cycle segments. In case the master is the data source (reset and write) the minimal values are optimal since they provide the fastest transfer rate. In case the slave is the data source (presence and read) the maximum values are optimal since they allow the signal to stabilize as much as possible.

Table 6: 1-wire protocol standard timing restrictions

timing [µs]	normal		overdrive		optimal	description
	min	max	min	max		
t_{RSTL}	480	640	48	80	min	reset low
t_{RSTH}	480	∞	48	∞	min	reset high (on newer devices this time is lower)
t_{RSTP}	60 +8.1	75	6 +1.3	10	max	reset presence detect (the extra delay is due to slow slave IO)
t_{DAT0}	60	120	6	16	min	data bit 0
t_{DAT1}	5	15	1	2	min	data bit 1
t_{DATS}		15		2	max	data sample
t_{REC}	5	∞	2	∞	min	recovery
t_{DLY}					1ms	idle delay (not standard, about t_{RSTL}+ t_{RSTH})
t_{RSTT}	960	∞	69	0		reset pulse cycle (t_{RSTL}+ t_{RSTH})
t_{DATT}	61	121	7	17		read/write slot cycle (t_{DAT0}+ t_{REC})

The exact timing of 1-wire cycles is provided as a table with 3 options for normal mode and 2 options for overdrive mode. For exact base time periods only typical values are provided, for range base time periods minimum and maximum value are provided.

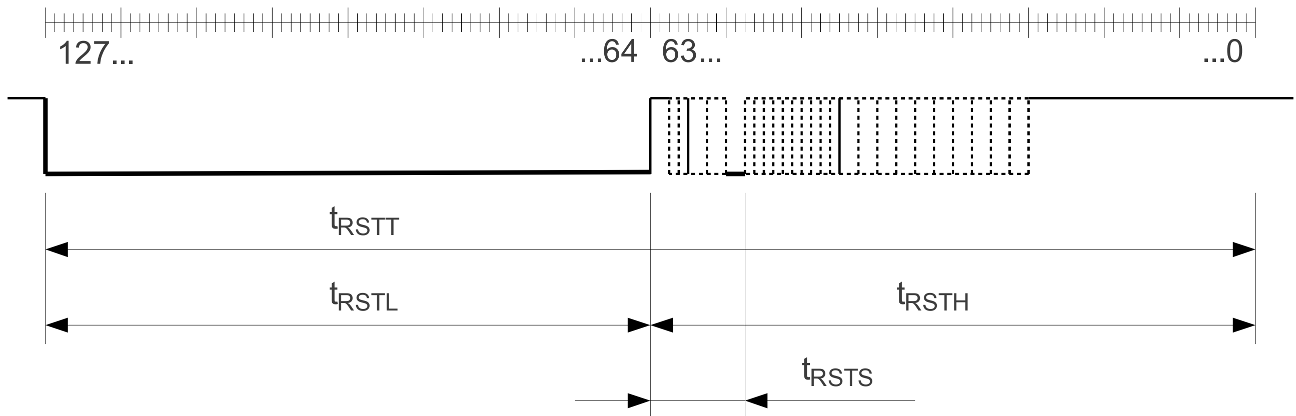
Table 7: Timing options

	normal mode times [µs]							overdrive mode times [µs]				
name	"7.5"		"5.0"		"6.0"			"1.0"		"0.5"		
timing	N	typ	N	typ	N	min	max	N	typ	N	min	max
base	1	7.5	1	5	1	6	7.5	1	1	1	0.5	0.66
t_{RSTL}	64	480	96	480	80	480	600	48	48	96	48	64
t_{RSTH}	64	480	96	480	80	480	600	48	48	96	48	64
t_{RSTP}	10	75	15	75	10	60	75	10	10	15	7.5	10
t_{DAT0}	8	60	12	60	10	60	75	6	6	12	6	8
t_{DAT1}	1	7.5	1	5	1	6	7.5	1	1	2	1	1.33
t_{DATS}	2	15	3	15	2	12	15	2	2	3	1.5	2
t_{REC}	1	7.5	1	5	1	6	7.5	2	2	4	2	2.66
t_{DLY}	128	960	200	1000	160	960	1200	96	96	192	96	128

The 7.5µs base time period option is also explained with timing diagrams.

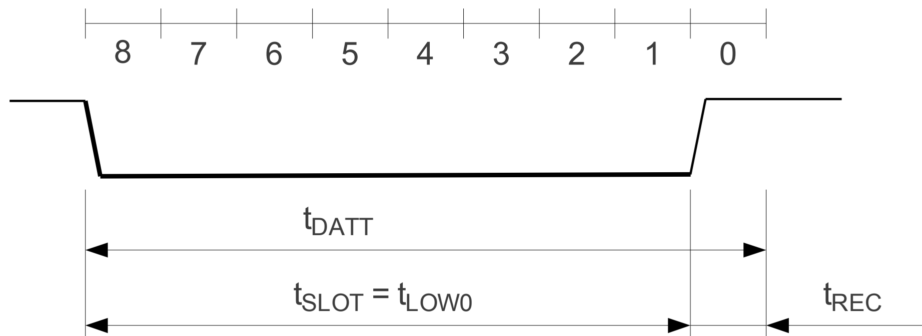
The reset cycle is 128 base periods long. Reset low and high time each take 64 base periods. After the rising edge on the wire a typical 1-wire slave would assert its presence after 4 base periods (min 2, max 8), and keep it asserted for 16 base periods (min 8, max 32). The presence status is sampled

when most stable at 10 base periods after the reset rising edge.



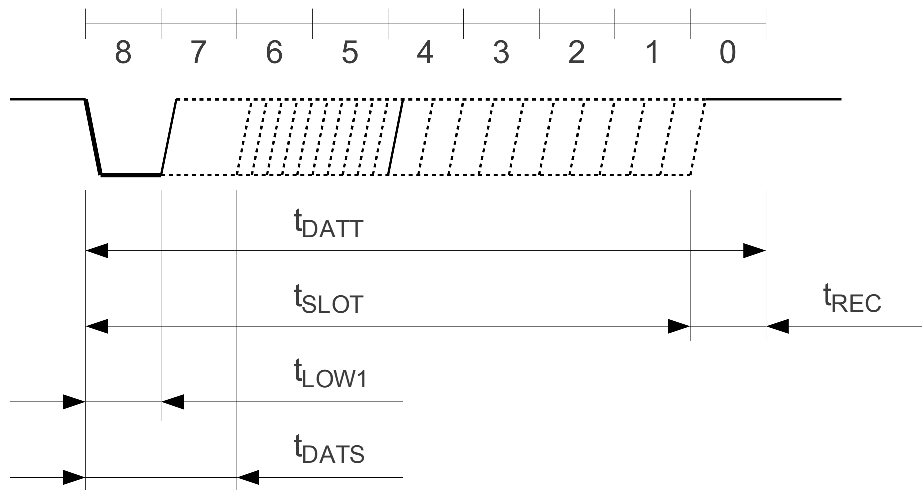
Drawing 4: Reset and presence timing diagram

The write '0' cycle is 9 base periods long. The wire is tight low for 8 base periods than released. The recovery time is 1 base period.



Drawing 5: Write 0 timing diagram

The write '1' or read cycle is 9 base periods long. The wire is tight low for 1 base period than released. A typical 1-wire slave would answer with '0' by holding the wire down for 4 base periods (min 2, max 8) from the falling edge of the read command. Data is sampled at the most stable point at 2 base periods from the falling edge. There is 1 base period at the end for recovery.



Drawing 7: Write 1 or read timing diagram

4 Processor interface

The processor interface is described in three steps:

1. address space
2. clock dividers
3. 1-wire line multiplexing and power enable functionality
4. control/status register description
5. driver access sequences (polling and interrupts)

The document is focused on describing the 32bit CPU interface. Some aspects of a 8bit interface are mentioned, but many details are omitted.

4.1 Address space

The sockit_own module is designed to be accessible by a 32bit CPU through a single 32bit register, if between 1 and 16 (`OWN` parameter) 1-wire lines are required. If a single line is enough than all but one power and all select bits have no function, and the interface can be compacted into an 8bit control/status register.

If implemented (`CDR_E=1`), there are two 16bit clock divider ratio registers (`cdr_n`, `cdr_o`) one for normal and one for overdrive mode.

Table 8: 32bit interface address space

<table>
<tr><th>address</th><th>31..24</th><th>23..16</th><th colspan="2">15..8</th><th>7..0</th></tr>
<tr><td>0</td><td colspan="2">power</td><td>unused</td><td>select</td><td>control/status</td></tr>
<tr><td>1</td><td colspan="2">cdr_o</td><td colspan="3">cdr_n</td></tr>
</table>

Since all the controls are in a single register and there is no byte select signal, on every write all controls must have the correct value. In software this can be achieved by having a permanent structure holding the interface current state.

Alternatively an 8bit interface is possible. The maximum number of 1-wire lines is reduced to 4 and the clock divider ratio register size is reduced to 8bit.

Table 9: 8bit interface address space

<table>
<tr><th>address</th><th colspan="3">7..0</th></tr>
<tr><td>0</td><td colspan="3">control/status</td></tr>
<tr><td>1</td><td>power</td><td>unused</td><td>select</td></tr>
<tr><td>2</td><td colspan="3">cdr_n</td></tr>
<tr><td>3</td><td colspan="3">cdr_o</td></tr>
</table>

4.2 Clock dividers

If clock divider ratio registers `cdr_n` and `cdr_o` are implemented (`CDR_E=1`) than the reset value of

the registers will be as defined by `CDR_N` and `CDR_O` parameters, and the registers can be written to with new divider ratios while there is no 1-wire cycle in progress. If the registers are not implemented (`CDR_E=0`) than a read will provide values defined by parameters and writes will be ignored.

4.3 1-wire line multiplexing and power enable

The sockit_owr module can implement more than one 1-wire line depending on the value of the `OWN` parameter. The main purpose of having multiple lines is the ability to attach sensors (thermometers for example) to separate lines, so that the physical position of the sensor can be deduced without requiring sensors with addressing or chain structures.

Since there is a single state counter in the module, it is not possible to communicate with more than one device at the same time. To be exact only one contemporary reset or write/read bit cycle can be issued, but it is possible to time multiplex between lines on bit transfer level. Each cycle must finish before a new cycle on the same or different line can be issued.

The `select[3:0]` register is used to select one from `OWN` lines. If the module is configured for only one line (`OWN=1`), than select is not implemented and therefore ignored. If a line is not selected its `owr_e` signal is not active. After reset line 0 is selected, but the proper value has to be provided on each write to the register.

The `power[15:0]` register is directly connected to the `owr_p[OWN-1:0]` output port. If the module is configured for only one line (`OWN=1`), than `power[15:0]` is not implemented and ignored and the `pwr` bit from the control/status register is used instead. If the module is configured for more than one line (`OWN=1`) than writes to the `pwr` bit are ignored. After reset all bits in the register are zero and all lines are only powered over the required weak pull-up resistor. By writing 1 into the n-th bit the n-th 1-wire line will provide power supply instead of being only an input. While power to a line is enabled it is not possible to perform 1-wire cycles on the same line, but it is possible to communicate on other lines.

4.4 Control/status register

The control/status register is used to initiate 1-wire cycles and to check their status.

Table 10: Control/status register structure

	7	6	5	4	3	2	1	0
name	`ien`	`irq`	`/`	`pwr`	`cyc`	`ovd`	`rst`	`dat`
access	r/w	r/c	/	r/w	r/w	r/w	r/w	r/w
reset	0	0	x	0	0	0	0	x

Table 11: Control/status register bits with descriptions

bit		description
0	`dat`	write: data transmit request (0 - perform write 0 slot, 1 - perform write 1 or/and read slot) read: data receive status (read value for each 1-wire read slot) read: presence detect status (0 - device presence detected, 1 - no device detected)
1	`rst`	reset pulse request (0 - data cycle, 1 - reset and read presence detect cycle is selected)

2	ovd	overdrive setting (0 - standard speed, 1 - overdrive speed)
3	cyc	write: initiate an 1-wire cycle (0 - no action, 1 - start a new cycle) read: 1-wire cycle status (0 - no cycle in progress, 1 - cycle in progress)
4	pwr	power supply (0 - normal operation, 1 - supply power over 1-wire)
5	/	reserved
6	irq	status of cycle interrupt (set on end of 1-wire cycle, cleared by bus read)
7	ien	enable cycle interrupt (0 - disabled, 1 - enabled)

Bits ovd, rst, dat are used to configure a cycle and bit cyc is used to start it. Bit dat provides the write data, bit rst selects a reset cycle and bit ovd is used to specify the transfer speed mode. Combinations of this three bits are used to select 8 different cycle types (see table below, timing values are for normal mode "5.0" and overdrive mode "1.0" base time periods). Detailed timing for this cycles is explained in the section "Supported 1-wire functionality".

Writing '1' into cyc register will always initiate an 1-wire cycle. Writing '1' into the register while a cycle is already in progress will break the running cycle and start a new cycle, usually causing a protocol error. When read, cyc shows the cycle status, which can be used for polling.

Table 12: List of supported cycles

ovd	rst	dat	timing	description
0	0	0	65µs	write '0' cycle
0	0	1	65µs	write '1' cycle or read cycle
0	1	0	960µs	reset and presence detect cycle
0	1	1	1000µs	timed idle cycle, can be used to generate timed delays
1	0	0	8µs	overdrive write '0' cycle
1	0	1	8µs	overdrive write '1' cycle or read cycle
1	1	0	96µs	overdrive reset and presence detect cycle
1	1	1	0µs	zero time idle cycle

Bit pwr has a different meaning for write than for read. The write functionality is only implemented if only a single 1-wire line is implemented (OWN=1), then the write value directly controls the status of the owr_p[0] signal. Read of pwr provides state of the selected line power (power[select] or pwr).

Read only bit irq describes the current interrupt request status and is set at the end of each 1-wire cycle. Interrupt status is cleared by a register read, so care must be taken to check its value on every read. Bit ien (interrupt enable) specifies if irq will trigger an interrupt or not.

4.5 Driver access sequences

The 1-wire master can be used to perform most actions defined in the iButton specification as the physical layer. It is possible to use polling or interrupts, both options will be described.

The reference implementation for Nios II HAL and µC/OS-II is available as source code.

The description uses the next pseudo functions:

- `write (data)` - 32bit (or 8bit) write to the configuration/status register
- `data = read ()` - 32bit (or 8bit) read from the configuration/status register

If clock divider ratio registers are implemented (`CDR_E=1`) they must be written with proper divider ratios prior to issuing any 1-wire cycles.

Note that the provided pseudo code does not implement power delivery and is designed for a single 1-wire lane (select is not used). For a proper implementation of this functionality see the ported public domain kit code (`owlnk.c`).

4.5.1 Polling routines

Polling is not an optimal implementation, but it might be useful for debugging purposes. Since interrupts are not used '0' should be written into interrupt enable register and cycle status bit `cyc` is used to identify the end of the cycle.

Two basic functions are provided one for reset pulse and the other for a combined write/read data slot. Separate write and read data functions are based on the combined data slot function.

Reset and presence pulses

A '1' is written into the `rst` bit and a '0' is written into the `dat` bit. Bit `ovd` is used to select normal or overdrive mode. The cycle will start after writing '1' into `cyc`. The software must wait in a loop till the cycle ends (`cyc=0`), and than the presence status can be read from bit `dat`. The next function returns the presence detect status.

```
bool owr_reset (bool overdrive)
{
  int reg;
  write (0x08 | (overdrive << 2) | 0x02);
  while (0x08 & (reg = read()));
  return (reg & 0x01);
}
```

Read/write data slots

A boolean value `bit` is written into the `dat` bit, this will cause a write `bit` cycle on the 1-wire bus. Bit `ovd` is used to select normal or overdrive mode. The cycle will start after writing '1' into `cyc`. The software must wait in a loop till the cycle ends (`cyc=0`), and than the read `bit` can be read from status register bit `dat`. The next function returns the read bit.

```
bool owr_slot (bool overdrive, bool bit)
{
  int reg;
  write (0x08 | (overdrive << 2) | bit);
  while (0x08 & (reg = read()));
  return (reg & 0x01);
}
```

Read data bit and write data bit functions are based on the slot function.

```
bool ovd_read  (bool overdrive)     {return owr_slot (overdrive, 0x1);}
void ovd_write (bool overdrive, bool bit)  {owr_slot (overdrive, bit);}
```

Delay of 0ms or 1ms

By writing '1' to `rst` and `dat` an idle cycle is produced. If `ovd` bit is set for normal mode the delay is of 1ms, if it is set for overdrive mode the delay is of zero time. The 1ms delay is intended to be used as a timed delay with power delivery, so the pseudo code provides the power option.

The next code generates a zero time delay.

```
void owr_delay_0 (bool power)
{
  write (0x08 | (power << 4) | 0x07);
  while (0x08 & read());
}
```

The next code generates an 1ms delay.

```
void owr_delay_1 (bool power)
{
  write (0x08 | (power << 4) | 0x03);
  while (0x08 & read());
}
```

4.5.2 Interrupt routines

The 1-wire protocol is slow compared to clock speeds of most embedded processors, even low power micro-controllers. Because of the low speed a polling driver might spend many clock periods (at leas 7 for bit read/write and 960 for the reset pulse) busy waiting for the 1-wire cycle to finish, this time could be better spent performing useful tasks. To be able to use this idle time, a task scheduler (usually provided by an operating system) is also required, so that there can be alternative threads that can be executed while waiting for the interrupt (or waiting for temperature conversion to finish).

Since the 1-wire bus can be accessed from more than one task, a locking mechanism is required, so that higher level protocol sequences are not corrupted. Only locking of low level transfers is described here. Locking for higher protocol layers is provided by the "public domain kit".

The provided pseudo code is for a pseudo OS, to implement the driver for a real OS, many OS specific details related to task and IO scheduling must be taken into account. The OS must provide the next functions (and others for initialization):

- `os_sem_lck (sem)` - lock the resource (it becomes unavailable to other tasks)
- `os_sem_rel (sem)` - release the resource lock
- `os_io_wait (flag)` - return the CPU to the OS and wait for IO data for `flag`
- `os_io_done (flag)` - tell the OS IO data for `flag` is available, this will end the wait

The main detail to be careful about is interrupt status bits being cleared by a status register read.

Reset and presence pulses, read/write data slots

The functions for reset pulses and data slots are similar to the polling versions. The busy wait while loop is replaced with the `os_io_wait` function that returns the CPU to the OS and waits for an IO event (interrupt) when the CPU is returned to the 1-wire driver. The 1-wire cycle is also wrapped into a semaphore based lock preventing a different task from disrupting the cycle.

```
bool owr_reset (bool overdrive)
{
  os_sem_lck (sem);
  write (0x80 | 0x08 | (overdrive << 2) | 0x02);
  os_flag_wait (flag);
  os_sem_rel (sem);
  reg = ;
  return (read() & 0x01);
}

bool owr_slot (bool overdrive, bool bit)
{
  os_sem_lck (sem);
  write (0x80 | 0x08 | (overdrive << 2) | bit);
  os_flag_wait (flag);
  os_sem_rel (sem);
  return (read() & 0x01);
}
```

OS initialization and interrupt service routine

The HAL or OS must declare and initialize the semaphore and flag and register the interrupt service routine.

```
os_type_sem  sem;
os_type_flag flag;

os_create_sem  (sem);
os_create_flag (flag);

os_irq_register (irq_sockit_owm);
```

The interrupt service routine must clear the interrupt (by reading the status register) and using the flag tell the OS that the wait for data has ended.

```
void irq_sockit_owm ()
{
  read();
  os_io_done (flag);
}
```

5 Software driver

A port of the Public Domain Kit (version 3.10r2) to Nios II HAL and µC/OS-II real time OS is available with the sockit_owr component. This section describes this port, and how it can be extended.

5.1 Port of Public Domain Kit components to Nios II HAL and µC/OS-II

The port is done based on instructions from Altera and Maxim.

5.1.1 Nios II HAL and µC/OS-II drivers

The integration into Nios II software development environment is done with a TCL script and C source and header files. The integration is done based on instructions from Altera.

Table 13: Files for Nios II HAL and µC/OS-II integration

file	description
`sockit_owm_hw.tcl`	TCL script for integration into SOPC Builder
`sockit_owm_sw.tcl`	TCL script for integration into Nios II EDS
`inc/sockit_owm_regs.h`	specification of hardware registers and low level access macros
`HAL/inc/sockit_owm.h`	initialization and interrupt handling code
`HAL/src/sockit_owm.c`	

The hardware TCL script provides the software with macros defining the hardware configuration. The software TCL script provides integration into the Nios II EDS, by defining:

- driver version and compatibility
- list of C sources and headers
- list of driver configuration options

The register definitions are mostly used for initialization and interrupts related code and for the 1-wire protocol link layer. The next IO macros are provided for reading from and writing into the control/status and clock divider ratio registers:

```
IORD_SOCKIT_OWM_CTL(base)
IOWR_SOCKIT_OWM_CTL(base, data)
IORD_SOCKIT_OWM_CDR(base)
IOWR_SOCKIT_OWM_CDR(base, data)
```

Interrupt support is provided for a system without or with the µC/OS-II real time OS. But there are no advantages using interrupts if there is no task scheduling implemented on the system. Since the sockit_owm module provides only bit sized data transfers, there are interrupts for each bit. If the CPU is running at low clock speeds, the operating system overhead might be substantial.

5.1.2 Public Domain Kit components

The Public Domain Kit provides a layered 1-wire protocol implementation. The focus of the port was on layers common to all 1-wire devices, device drivers with the exception of thermometers

were omitted. Support for other devices and applications have to be ported separately by the user.

The ported files were located in the next directories from the Public Domain Kit package:

- `owpd310r2/common` - some common code (CRC, err, findtype) and device drivers
- `owpd310r2/apps/temp` - the thermometer demo application is from here
- `owpd310r2/doc` - documentation
- `owpd310r2/lib/general` - data link and session layer TODO files
- `owpd310r2/lib/general/shared` - transport and network layer files

Table 14: Files from the Public Domain Kit

file	description
`HAL/inc/ownet.h`	main protocol stack header file
`HAL/src/owlnk.c`	1-wire protocol data link layer functions
`HAL/inc/ownet.c`	1-wire protocol network layer functions
`HAL/src/owtran.c`	1-wire protocol transport layer functions
`HAL/src/owses.c`	1-wire protocol session layer functions
`HAL/src/owerr.c`	1-wire driver error handling
`HAL/src/crcutil.c`	1-wire protocol CRC utilities
`HAL/inc/findtype.[hc]`	1-wire network search for devices
`HAL/inc/temp10.[hc]`	thermometer device driver (ID 0x10 - DS1920/DS1820)
`HAL/inc/temp28.[hc]`	thermometer device driver (ID 0x28 - DS18B20)
`HAL/inc/temp42.[hc]`	thermometer device driver (ID 0x42 - DS28EA00)

Most code changes were done to `owlnk.c` and `owses.c`. Some platform specific code for unrelated platforms was removed from `ownet` and `owerr`. Other files are mostly unchanged, check the Git repository to see details.

5.2 Adding support for new devices

Support for a new device can be added by copying needed files from the `/common` directory in the Public Domain Kit into the working directory of the Nios II project. Applications can also be ported from the `/apps` directory.

5.3 Possible improvements

The main limitation the user has to be careful with is only one hardware module instance can be handled by the driver. This is due to how the Public Domain Kit handles ports (1-wire lines) and its use of global variables. Since this affects almost all high level functions in the Public Domain Kit, there would be too many changes necessary to enable multiple instances.

So a possible improvement would be to rewrite the Public Domain Kit to have a structure instead of an integer for the `portnum` variable, this structure would also store global variables.

6 Altera development tools integration

The `sockit_owm` component can be used in any FPGA, but special care has been taken integrating it into the Altera development tools environment. The directory `sockit_owm` can be used as a portable component for easy integration into an Altera FPGA design.

6.1 SOPC Builder and Nios II EDS integration

Integration is done according to Altera specification with two TCL scripts.

- `sockit_owm_hw.tcl` for SOPC Builder integration
- `sockit_owm_sw.tcl` for Nios II EDS integration

6.1.1 SOPC Builder

Adding the component to a project

SOPC builder component IP search path must point to the directory containing the `sockit_owm` directory. The easiest way to achieve this is to put `sockit_owm` into the directory of the Quartus II project and add the project directory to the search path. Use this menu path:

SOPC Builder > Tools > Options > IP Search Path > Add ...

Once the proper IP search path is set, the module will be available under:

Library > Interface Protocols > Serial > 1-wire (onewire)

Configuring the component

The next configuration options are available in the component GUI:

- `OVD_E` enable overdrive mode implementation
- `CDR_E` enable clock divider ratio register implementation
- `OWN` select the number of implemented 1-wire lines
- `BTP_N` base time period for normal mode
- `BTP_O` base time period for overdrive mode

Overdrive mode can be disabled (`OVD_E=0`) in case devices that only support normal mode will be used. Disabling overdrive mode implementation provides a small reduction in logic consumption.

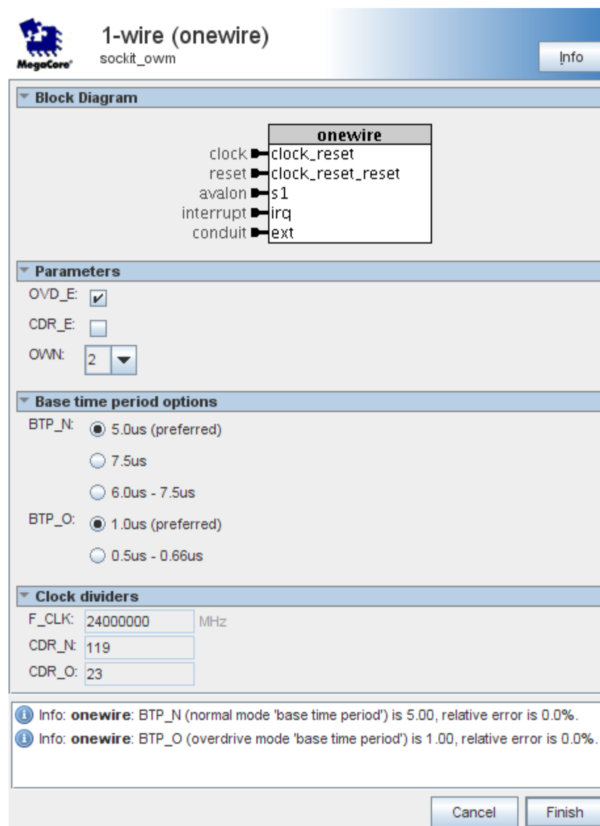
For low-end and midrange devices the clock frequency is usually defined at compile and fixed during runtime, so there is no need for software to change values of clock divider ratio registers. If the registers are not implemented (`CDR_E=0`) the divider values are constant. Disabling clock divider ratio registers implementation provides a small reduction in logic consumption.

The number of 1-wire lines can be set with `OWN`. Although many 1-wire devices can be connected to the same line there are usage scenarios where the device can be identified by the line it is connected to. An example would be more than one thermometer on the same circuit board.

It is possible to change the base time period for normal and overdrive mode, although the default is usually appropriate.

The TCL script calculates the correct clock divider ratios automatically based on the system clock

frequency and base time periods. The script also reports deviations from ideal values and if the error is more than 2% an error event is generated.



Drawing 8: Component GUI for sockit_owm

6.1.2 Nios II EDS

The C driver is integrated into the BSP automatically if a SOPC Builder project containing the `sockit_owm` component is used to generate the BSP.

The system requires some application code to use the driver. An example application is present in the demo. Device driver files from the public domain kit can be added to the project and device drivers already included in the base driver can be disabled within BSP configuration tools.

The user can configure the driver with some options, but the default values should be appropriate for most projects. This options are translated into define macros and can be enabled or disabled within BSP configuration tools.

Table 15: Driver configuration options

option	default	description
`SOCKIT_OWM_POLLING`	disabled	use polling instead of interrupts
`SOCKIT_OWM_HW_DLY`	enabled	use the hardware delay cycle instead of `usleep`
`SOCKIT_OWM_ERR_ENABLE`	enabled	enable the error code from the public domain kit
`SOCKIT_OWM_ERR_SMALL`	enabled	small version of the error implementation

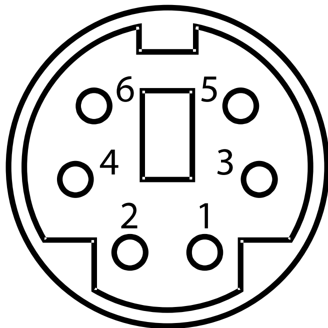
7 Demo hardware and software implementation

The popular Terasic DE1 demo board is used for a demo implementation. 1-wire devices can be connected directly to the PS2 port. The demo software is using the JTAG UART to communicate with the user.

This section also provides hardware logic an software memory sizes.

7.1 Demo hardware

The demo project is a Nios II SoC with 8MiB SDRAM for software and JTAG UART for communication with the Nios II EDS. Two 1-wire lines are connected to the PS/2 connector.



Drawing 9: Image of the PS/2 connector on the Terasic DE1 board

The PS/2 connector is used since it provides a 5V power supply and the signals already have 2kΩ pull-up resistors (required to implement the 1-wire bus) and 120Ω series resistors (useful for preventing damage to the board). The DATA and CLK pins are used to connect the two 1-wire lines.

Table 16: Pin-out of the PS/2 connector on the Terasic DE1 board

pin	name	description
1	`OWR[0]`	1-wire line 0 with a 2kΩ pull-up to 5V and 120Ω serial resistor to the FPGA pin
2	`NC`	not connected
3	`GND`	ground
4	`VCC`	+5V power supply
5	`OWR[0]`	1-wire line 1 with a 2kΩ pull-up to 5V and 120Ω serial resistor to the FPGA pin
6	`NC`	not connected

1-wire devices can be connected to the two lines in any combination. A single DS1820 or DS18B20 thermometer in TO92 package can even be directly inserted into the PS/2 connector.

7.1.1 Demo hardware logic consumption

The demo hardware has the default parameters ("5.0" and "1.0" base time periods, no clock divider registers) except for the two 1-wire lines. This configuration is implemented inside an Altera Cyclone II device running at 48MHz and consumes 66 combinatorial LC and 26 LC registers.

To fit inside smaller devices like CPLDs with few logic cells the next steps can be taken:

- implement a single lane, so select and power registers will not be implemented
- lower the clock frequency, so a smaller clock divider is required
- disable overdrive mode
- use the 8bit bus width option
- fix the bus address to 0, so read of clock divider ratios will not be implemented
- use the "7.5" base time period, this should reduce the state machine size

7.2 Demo software

Two options for the demo software are available, one running directly on the Nios II HAL, the other requires µC/OS-II.

Table 17: Demo software main function files

fail with main function	description
`demo/Terasic_DE1/software/onewire/onewire.c`	Nios II HAL main
`demo/Terasic_DE1/software/onewire_ucosii/onewire_ucosii.c`	µC/OS-II main

To run the demo software an appropriate BSP must be generated (with or without µC/OS-II). An UART is used to show the software progress. The demo focuses on thermometers, it will detect any thermometer within one of the three device groups 0x10, 0x28 and 0x42. The demo will display the device unique ID and the temperature measurement. The next printout is the result of running the demo with two thermometers (DS18S20 and DS18B20).

```
Temperature device demo:

(0) 3F000000C8CF9B28  22.3 Celsius
(1) 44000801E51EC510  22.8 Celsius

Press any key to continue
```

7.2.1 Demo software memory consumption

An estimate of 10kB for code size is based on the thermometer demo, memory consumption can be reduced by removing device drivers or unused protocol layers. The memory utilization by file can be found in `project_name.map`.

```
0x25c ../onewire_bsp/libhal_bsp.a(owerr.o)
0x2cc ../onewire_bsp/libhal_bsp.a(crcutil.o)
0x0d0 ../onewire_bsp/libhal_bsp.a(findtype.o)
0x710 ../onewire_bsp/libhal_bsp.a(owlnk.o)
0xdbc ../onewire_bsp/libhal_bsp.a(ownet.o)
0x110 ../onewire_bsp/libhal_bsp.a(owses.o)
0x37c ../onewire_bsp/libhal_bsp.a(owtran.o)
0x0c0 ../onewire_bsp/libhal_bsp.a(sockit_owm.o)
0x224 obj/temp10.o
0x2b4 obj/onewire.o (main)
```

8 Testing

The sockit_owm component was tested on two levels. The Verilog RTL was tested in simulation and the C driver was tested on the demo implementation.

8.1 Verilog RTL simulation

Extensive RTL simulation testing is required, since the RTL module can be used separately from the provided C driver (with a Linux kernel driver for example) and can be implemented in an ASIC.

Table 18: Verilog RTL testbench files

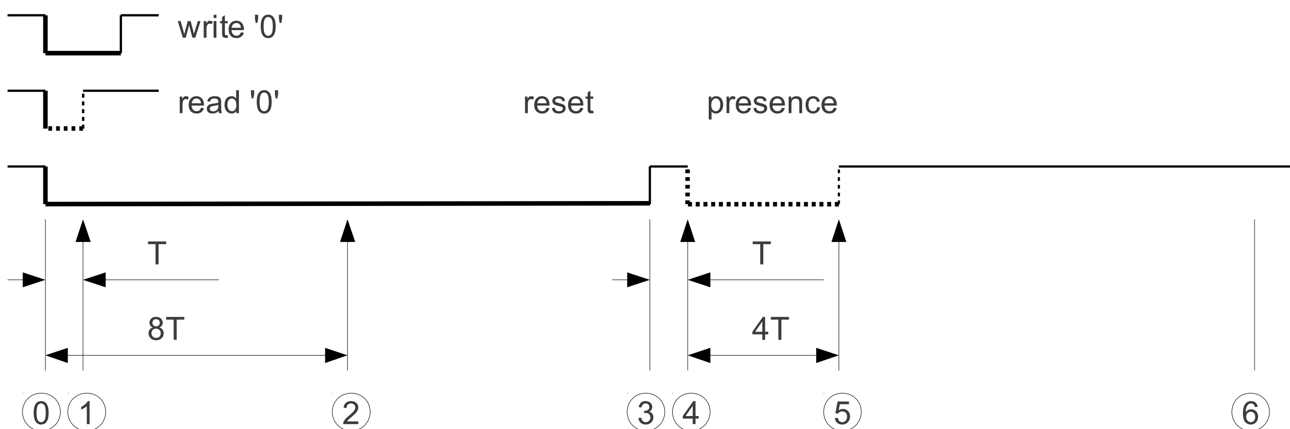
file	description
`hdl/onewire_slave_model.v`	1-wire slave with parameterizable timing and debug features
`hdl/onewire_tb.v`	Verilog testbench
`sim/iverilog_gtkwave.scr`	Bash script for running the bench with RTL parameter options
`sim/iverilog_gtkwave.cmd`	Windows cmd script
`sim/gtkwave.sav`	GTKWave waveform save file

8.1.1 1-wire slave model

The Verilog 1-wire slave model `onewire_slave_model.v` is based on the assumption that 1-wire devices have an internal time base (probably implemented with a RC oscillator) on which an internal state machine is running.

For normal mode this time base is typically T_{typ}=30µs, but can vary from a minimum of $T_{min}=T_{typ}/2$=15µs on fast devices to a maximum of $T_{max}=T_{typ}*2$=60µs on slow devices. This assumptions are based on the the 1-wire protocol specification mentioning that the fastest device is 4 times faster than the slowest device.

For overdrive mode the typical time base seems to be $T_{typ,o}=T_{typ}/8$=3.75µs. The tolerances for devices supporting overdrive mode seem to be more strict, and on the fast side. So for fast devices $T_{min,o}$≈16µs/8=2µs and for slow devices T_{max}≈48µs/8=6µs.



Drawing 10: 1-wire slave model timing diagram

1-wire slave model timing diagram check points are assumptions based on the 1-wire timing from the standard.

`0` - the 1-wire master pulls the line low

for a read slot the slave pulls the line low if read data is '0'

`1` - for a write slot the slave samples data from the line state

for a read '0' slot the slave releases the line

`2` - the slave samples reset status from the line state

`3` - the master releases the line at the end of reset low time

`4` - the slave pulls the line low to indicate its presence

`5` - the slave releases the line, ending presence indication

8.1.2 Verilog testbench

The bench code is separated into two parts, a Verilog testbench for RTL internals and a Bash script for parameters. The Verilog testbench tests the RTL logic and 1-wire timing. The logic tests are:

- CPU bus access
- clock dividers
- 1-wire state machine
- overdrive mode
- power delivery

1-wire timing tests are performed on three 1-wire slave models with different timings: minimum (2 times faster than typical), typical and maximum (2 times slower than typical). This should ensure that the timing from the protocol are implemented correctly. The next 1-wire cycles are tested:

- reset pulse
- write/read '0'/'1'
- 1ms delay pulse
- 0ms idle pulse

With each test data transfers in both directions are checked (presence detection, read from slave, write to slave) and power delivery status is also checked.

8.1.3 Parameter tests

A Bash script is used to run the testbench with different parameters:

- 32 and 8 bit CPU data bus width
- clock divider ratio registers enabled/disabled
- base time period normal/overdrive mode option pairs ("5.0", "1.0"), ("6.0", 0.5") and ("7.5", overdrive not implemented)

The Bash script can be modified to run a specific test. The parameter nested loop can be commented out and the last test will generate waveforms and open them in GTKWave.

8.2 C driver tests

C drivers are tested using the demo implementation. Most of the tests focus on 1-wire functionality, but some focus on Nios II HAL and μC/OS-II.

The next 1-wire functionality was tested:

- functions reset pulse, presence detection, read/write slots
- overdrive mode
- power delivery

The next parts of Nios II HAL integration were tested:

- hardware 1ms delay
- software 1ms delay (it is longer than it should be)
- polling driver
- interrupt driven driver

The next parts of μC/OS-II integration were tested:

- accessing the driver from two tasks simultaneously
- returning the CPU to the OS while a task is waiting for an 1-wire cycle to complete

8.3 Testing TODO

Not everything was tested and some parts might not be properly implemented:

- timed delay functions provided by μC/OS-II (not implemented properly)
- semaphore protection of 1-wire cycles (probably works, but was not tested)
- 1-wire devices other than thermometers

9 License

Verilog RTL and testbench files are licensed under LGPL 3.

Bash and Windows cmd scripts are licensed under LGPL 3.

http://www.gnu.org/licenses/lgpl.html

TCL scripts for integration into Altera tools are licensed under LGPL 3. This files were at first based on Altera demos, but are now entirely rewritten according requirements described in SOPC Builder documentation.

http://www.gnu.org/licenses/lgpl.html

The glue code for the Nios II HAL is licensed with the original public domain like license by Altera. This files were at first based on Altera demos, but are now almost entirely rewritten according requirements described in Nios II documentation.

The license for the "public domain kit" has not been changed. The modified code is also public domain.

lic/license_pub.txt

The documentation (including all drawings except the PS/2 connector) is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) License.

http://creativecommons.org/licenses/by-sa/3.0/

http://creativecommons.org/licenses/by-sa/3.0/legalcode

The PS/2 connector SVG drawing is from Wikipedia, and its copyright holder released it into the public domain.

http://en.wikipedia.org/wiki/File:MiniDIN-6_Connector_Pinout.svg

10 References

1-Wire Products Design Guide

http://www.maxim-ic.com/design_guides/en/1_WIRE_PRODUCTS_4.pdf

iButton Overview

http://pdfserv.maxim-ic.com/en/an/appibstd.pdf

1-Wire Public Domain Kit

http://files.dalsemi.com/auto_id/public/owpd310r2.zip

http://pdfserv.maxim-ic.com/en/an/AN1097.pdf

http://pdfserv.maxim-ic.com/en/an/AN155.pdf

DS18S20 1-Wire Parasite-Power Digital Thermometer

http://datasheets.maxim-ic.com/en/ds/DS18S20.pdf

DS18B20 Programmable Resolution 1-Wire Digital Thermometer

http://datasheets.maxim-ic.com/en/ds/DS18B20.pdf

SOPC Builder User Guide

4. SOPC Builder Components

7. Component Interface Tcl Reference

10. SOPC Builder Component Development Walkthrough

http://www.altera.com/literature/ug/ug_sopc_builder.pdf

Avalon MM

www.altera.com/literature/manual/mnl_avalon_spec.pdf

µC/OS-II

Jean J. Labrosse: MicroC OS II: The Real Time Kernel