stecman / **Makefile**

Last active 2 months ago

DS18B20 1-Wire implementation for Atmel AVR

_DS18B20-AVR-OneWire.md

AVR C 1-Wire + DS18B20 temperature sensor

This implements Maxim's One Wire / 1-Wire protocol for AVR microcontrollers from the DS18B20 datasheet and Maxim's other application notes. The implementation should be generic enough that it can be adapted to other microcontrollers.

Usage

Single DS18B20 device on bus

With just one DS18B20 on the 1-Wire bus, you can ignore slave addresses:

```
#include "pindef.h"
#include "onewire.h"
#include "ds18b20.h"

// ...

// This is a hacked together interface to pass around a port/pin combination by refe
// Everything in pindef.h compiles to quite inefficient asm currently.
const gpin_t sensorPin = { &PORTD, &PIND, &DDRD, PD3 };
```

```
// Send a reset pulse. If we get a reply, read the sensor
if (onewire_reset(&sensorPin)) {

    // Start a temperature reading (this includes skiprom)
    ds18b20_convert(&sensorPin);

    // Wait for measurement to finish (750ms for 12-bit value)
    _delay_ms(750);

    // Get the raw 2-byte temperature reading
    int16_t reading = ds18b20_read_single(&sensorPin);

    if (reading != kDS18B20_CrcCheckFailed) {
        // Convert to floating point (or keep as a Q12.4 fixed point value)
        float temperature = ((float) reading) / 16;
    } else {
        // Handle bad temperature reading CRC
        // The datasheet suggests to just try reading again
    }
}
```

Multiple devices on the bus

With more than one DS18B20 on the bus, you'll need to query the unique 64-bit address of each device and address them directly when sending other commands.

```
#include "onewire.h"

// ...

// Prepare a new device search
onewire_search_state search;
onewire_search_init(&search);

// Search and dump temperatures until we stop finding devices
while (onewire_search(&sensorPin, &search)) {
    if (!onewire_check_rom_crc(&search)) {
        // Handle ROM CRC check failed
        continue;
    }

    // Do something with the address
    // eg. memcpy(addresses[slaveIndex], search.address, 8);
}
```

To send commands to a specific slave, use `onewire_match_rom` instead of `onewire_skiprom`:

```
  onewire_match_rom(&sensorPin, address);
```

Search ROM implementation

I found the Search ROM flowchart in Maxim's datasheets and application notes a little hard to follow. Below is a diagram I threw together to maintain sanity implementing this. It goes with p51-54 from Maxim's application note 937: Book of iButton® Standards.

Also see application note 187: 1-Wire Search Algorithm. The implementation described there adds additional complexity/functionality to the search process, but also includes a (fairly verbose) implementation in C.

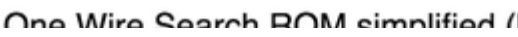
Example implementation

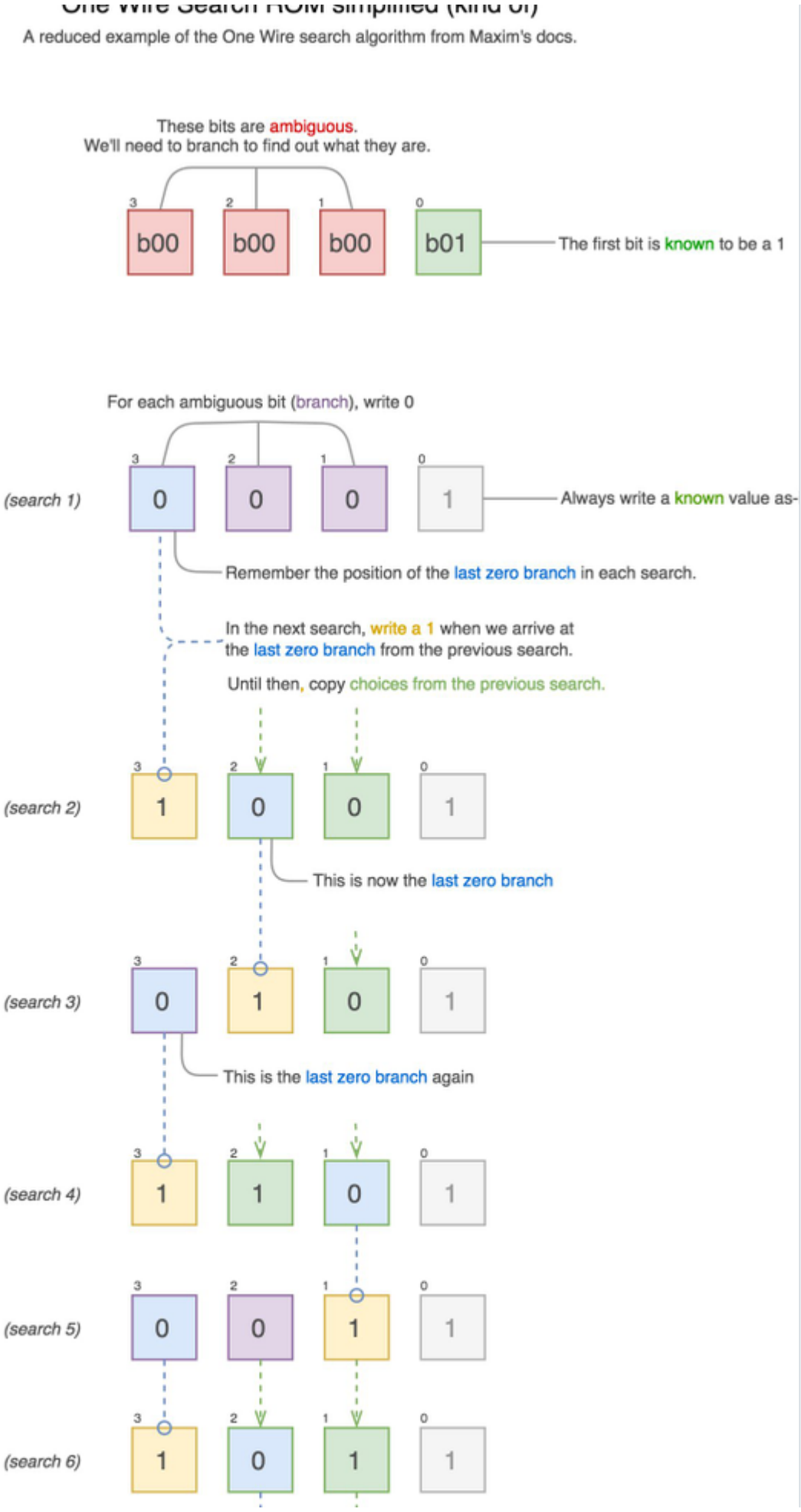
An example that triggers a conversion for all devices on the bus, then prints the reading from each individually is contained in `example.c` and can be built with the included `Makefile`.

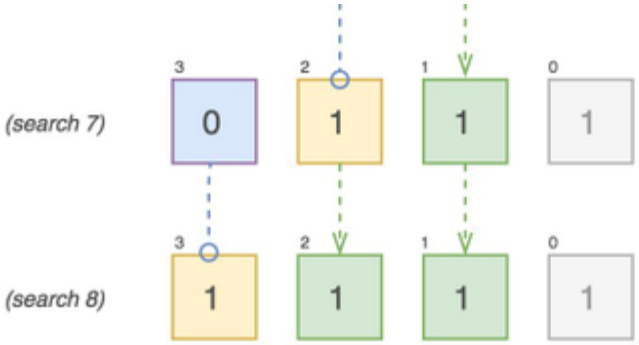
This is targeted at the ATmega328P. The serial printing implementation may need tweaking to work on other AVRs.

```
# Grab the latest version of this Gist
git clone https://gist.github.com/9ec74de5e8a5c3c6341c791d9c233adc.git avrc-ds18b20-
cd avrc-ds18b20-onewire

# Build (requires gcc-avr and avr-libc)
make

# Flash (requires avrdude)
# You may need to tweak the PROGRAMMER variable at the top of the Makefile to work w
# This is currently set to program over serial using the Arduino bootloader on /dev/
make flash
```

_Search-ROM-diagram.png




One Wire Search ROM simplified (kind of)
A reduced example of the One Wire search algorithm from Maxim's docs.
These bits are ambiguous.
We'll need to branch to find out what they are.
3 2 1 0
b00 b00 b00 b01
The first bit is known to be a 1
For each ambiguous bit (branch), write 0
(search 1)
0 0 0 1
Always write a known value as-
Remember the position of the last zero branch in each search.
In the next search, write a 1 when we arrive at
the last zero branch from the previous search.
Until then, copy choices from the previous search.
(search 2)
1 0 0 1
This is now the last zero branch
(search 3)
0 1 0 1
This is the last zero branch again
(search 4)
1 1 0 1
(search 5)
0 0 1 1
(search 6)
1 0 1 1



crc.c

```c
#include "crc.h"

// AVR
#include <util/crc16.h>

uint8_t crc8(uint8_t* data, uint8_t len)
{
    uint8_t crc = 0;

    for (uint8_t i = 0; i < len; ++i) {
        crc = _crc_ibutton_update(crc, data[i]);
    }

    return crc;
}
```

crc.h

```c
// C
#include <stdint.h>

/**
 * Calculate the CRC8 for an array of bytes
 *
 * This uses the polynomial (x^8 + x^5 + x^4 + 1)
 *
 * @returns the computed CRC8
 */
uint8_t crc8(uint8_t* data, uint8_t len);
```

ds18b20.c

```c
#include "ds18b20.h"
#include "crc.h"
#include "onewire.h"

```

```
// AVR
#include <util/delay.h>

// Command bytes
static const uint8_t kConvertCommand = 0x44;
static const uint8_t kReadScatchPad = 0xBE;

// Scratch pad data indexes
static const uint8_t kScratchPad_tempLSB = 0;
static const uint8_t kScratchPad_tempMSB = 1;
static const uint8_t kScratchPad_crc = 8;

static uint16_t ds18b20_readScratchPad(const gpin_t* io)
{
    // Read scratchpad into buffer (LSB byte first)
    static const int8_t kScratchPadLength = 9;
    uint8_t buffer[kScratchPadLength];

    for (int8_t i = 0; i < kScratchPadLength; ++i) {
        buffer[i] = onewire_read(io);
    }

    // Check the CRC (9th byte) against the 8 bytes of data
    if (crc8(buffer, 8) != buffer[kScratchPad_crc]) {
        return kDS18B20_CrcCheckFailed;
    }

    // Return the raw 9 to 12-bit temperature value
    return (buffer[kScratchPad_tempMSB] << 8) | buffer[kScratchPad_tempLSB];
}

uint16_t ds18b20_read_single(const gpin_t* io)
{
    // Confirm the device is still alive. Abort if no reply
    if (!onewire_reset(io)) {
        return kDS18B20_DeviceNotFound;
    }

    // Reading a single device, so skip sending a device address
    onewire_skiprom(io);
    onewire_write(io, kReadScatchPad);

    // Read the data from the scratch pad
    return ds18b20_readScratchPad(io);
}

uint16_t ds18b20_read_slave(const gpin_t* io, uint8_t* address)
{
    // Confirm the device is still alive. Abort if no reply
```

```
    if (!onewire_reset(io)) {
        return kDS18B20_DeviceNotFound;
    }

    onewire_match_rom(io, address);
    onewire_write(io, kReadScatchPad);

    // Read the data from the scratch pad
    return ds18b20_readScratchPad(io);
}

void ds18b20_convert(const gpin_t* io)
{
    // Send convert command to all devices (this has no response)
    onewire_skiprom(io);
    onewire_write(io, kConvertCommand);
}
```

ds18b20.h

```
#pragma once

#include "pindef.h"

// C
#include <stdint.h>

// Special return values
static const uint16_t kDS18B20_DeviceNotFound = 0xA800;
static const uint16_t kDS18B20_CrcCheckFailed = 0x5000;

/**
 * Trigger all devices on the bus to perform a temperature reading
 * This returns immedidately, but callers must wait for conversion on slaves (max 750ms)
 */
void ds18b20_convert(const gpin_t* io);

/**
 * Read the last temperature conversion from the only probe on the bus
 *
 * If there is a single slave on the one-wire bus the temperature data can be
 * retrieved without scanning for and targeting device addresses.
 *
 * Calling this with more than one slave on the bus will cause data collision.
 */
uint16_t ds18b20_read_single(const gpin_t* io);

/**
 * Read the last temperature conversion from a specific probe
```

```
 * Address must be a an array of 8 bytes (uint8_t[8])
 */
uint16_t ds18b20_read_slave(const gpin_t* io, uint8_t* address);
```

example.c

```
#include "ds18b20.h"
#include "onewire.h"
#include "pindef.h"

// AVR
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

// C
#include <stdint.h>
#include <stdlib.h>
#include <string.h>


/**
 * Set up the ATmega328P's UART peripheral to transmit
 *
 * The contents of this may need to be changed if you're using a different chip
 * or a different CPU frequency.
 */
void init_usart(void)
{
    cli();

    // Set baud rate to 9600 (with 16MHz clock)
        // (See "Examples of Baud Rate Setting" in the ATmega328P datasheet)
    UBRR0H = (uint8_t)(103 >> 8);
    UBRR0L = (uint8_t)(103);

    // Enable receiver and transmitter
    UCSR0B = _BV(RXEN0) | _BV(TXEN0);

    // Set frame format: 8 data, 1 stop bit
    UCSR0C = (3<<UCSZ00);
    UCSR0C &= ~(1<<USBS0);

    sei();
}

/**
 * Send a single byte out over serial (blocking)
 */
```

```
44  void usart_transmit(uint8_t data)
45  {
46      // Wait for empty transmit buffer
47      while ( !( UCSR0A & (1<<UDRE0)) );
48
49      // Put data into buffer, sends the data
50      UDR0 = data;
51  }
52
53  /**
54   * Block until a single byte is received over serial
55   */
56  uint8_t usart_receive(void)
57  {
58      // Wait for data to be received
59      while ( !(UCSR0A & (1<<RXC0)) );
60
61      // Get and return received data from buffer
62      return UDR0;
63  }
64
65  /**
66   * Print a null-terminated string
67   */
68  void print(char* string)
69  {
70      uint8_t i = 0;
71      while (string[i] != '\0') {
72          usart_transmit(string[i]);
73          ++i;
74      }
75  }
76
77  /**
78   * Print a null-terminated string followed by an automatic line break
79   */
80  void println(char* string)
81  {
82      print(string);
83      usart_transmit('\n');
84  }
85
86  /**
87   * Convert a fixed-point 12.4 number into a floating point string
88   * This is creates a human-readable string for temperature values from a DS18B20
89   *
90   * The passed buffer must be able to accomodate up to 9 characters plus a null terminator.
91   */
92  void fptoa(uint16_t value, char* buf, uint8_t bufsize)
```

```
{
    const uint16_t integer = (value >> 4);
    const uint16_t decimal = (value & 0x0F) * 625; // One 16th is 0.625

    memset(buf, '\0', bufsize);

    // Convert the integer portion of the number to a string
    itoa(integer, buf, 10);

    // Add decimal point
    buf += strlen((char*)buf);
    (*buf++) = '.';

    if (decimal == 0) {
        // Fill zeros to maintain fixed decimal places
        memset(buf, '0', 4);
        return;
    }

    if (decimal < 1000) {
        // Add a leading zero for .0625
        (*buf++) = '0';
    }

    // Convert the decimal portion of the number to a string
    itoa(decimal, buf, 10);
}

const static gpin_t sensorPin = { &PORTD, &PIND, &DDRD, PD3 };

int main(void)
{
    init_usart();

    for (;;) {
        if (!onewire_reset(&sensorPin)) {
            println("Nothing on the bus?");
            _delay_ms(1000);
            continue;
        }

        // Intiate conversion and wait for it to finish
        ds18b20_convert(&sensorPin);
        _delay_ms(750);

        // Prepare a new device search
        onewire_search_state search;
        onewire_search_init(&search);

```

```
        // Search and dump temperatures until we stop finding devices
        while (onewire_search(&sensorPin, &search)) {
            if (!onewire_check_rom_crc(&search)) {
                print("Bad ROM CRC");
                continue;
            }

            uint16_t reading = ds18b20_read_slave(&sensorPin, search.address);

            if (reading != kDS18B20_CrcCheckFailed) {
                // You can get the temperature as floating point degrees if needed, though
                // working with floats on small AVRs eats up a lot of cycles and code space:
                //
                //   float temperature = ((float) reading) / 16;
                //

                // Convert fixed-point to a printable string
                char buf[10];
                fptoa(reading, buf, sizeof(buf));
                print(buf);

            } else {
                println("Bad temp CRC");
            }

            print("  ");
        }

        print("\n");
    }

    return 0;
}
```

Makefile

```
DEVICE = atmega328p
CLOCK  = 16000000
PROGRAMMER = -c arduino -P /dev/ttyUSB0 -b57600

AVRDUDE = avrdude $(PROGRAMMER) -p $(DEVICE)

SOURCES = $(shell find . -name '*.c')
OBJECTS = $(SOURCES:.c=.o)

# Automatic dependency resolution
DEPDIR := .d
$(shell mkdir -p $(DEPDIR) >/dev/null)
DEPFLAGS = -MT $@ -MMD -MP -MF $(DEPDIR)/$*.Td
```

```
14
15  # Compiler flags
16  CFLAGS = -Wall -Os -DF_CPU=$(CLOCK) -mmcu=$(DEVICE)
17  CFLAGS += -I -I. -I$(shell pwd)
18  CFLAGS += -funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums
19  CFLAGS += -ffunction-sections -fdata-sections -Wl,--gc-sections
20  CFLAGS += -Wl,--relax -mcall-prologues
21  CFLAGS += -std=gnu11 -Wstrict-prototypes
22
23  # Specfic warnings as errors
24  CFLAGS += -Werror=return-type
25
26  # Enable coloured output from avr-gcc
27  CFLAGS += -fdiagnostics-color=always
28
29  COMPILE = avr-gcc $(DEPFLAGS) $(CFLAGS)
30
31  POSTCOMPILE = @mv -f $(DEPDIR)/$*.Td $(DEPDIR)/$*.d && touch $@
32
33  # symbolic targets:
34  all: $(SOURCES) main.hex
35
36  .c.o: $(DEPDIR)/%.d
37          @mkdir -p "$(DEPDIR)/$(shell dirname $@)"
38          $(COMPILE) -c $< -o $@
39          $(POSTCOMPILE)
40
41  .S.o:
42          $(COMPILE) -x assembler-with-cpp -c $< -o $@
43          # "-x assembler-with-cpp" should not be necessary since this is the default
44          # file type for the .S (with capital S) extension. However, upper case
45          # characters are not always preserved on Windows. To ensure WinAVR
46          # compatibility define the file type manually.
47
48  .c.s:
49          $(COMPILE) -S $< -o $@
50
51  flash: all
52          $(AVRDUDE) -U flash:w:main.hex:i
53
54  # Xcode uses the Makefile targets "", "clean" and "install"
55  install: flash
56
57  clean:
58          # Remove intermediates
59          find . -name '*.d' -or -name '*.o' -exec rm {} +
60
61          # Remove binaries
62          rm -f main.hex main.elf
```

```
63
64          # Remove dependency tracking files
65          rm -rf "$(DEPDIR)"
66
67  main.elf: $(OBJECTS)
68          $(COMPILE) -o main.elf $(OBJECTS)
69
70  main.hex: main.elf
71          rm -f main.hex
72          avr-objcopy -j .text -j .data -O ihex main.elf main.hex
73          avr-size --format=avr --mcu=$(DEVICE) main.elf
74
75  disasm: main.elf
76          avr-objdump -d main.elf
77
78  $(DEPDIR)/%.d: ;
79  .PRECIOUS: $(DEPDIR)/%.d
80
81  include $(wildcard $(patsubst %,$(DEPDIR)/%.d,$(basename $(SOURCES))))
```

onewire.c

```
1   #include "onewire.h"
2   #include "crc.h"
3
4   #include <util/delay.h>
5
6   bool onewire_reset(const gpin_t* io)
7   {
8       // Configure for output
9       gset_output_high(io);
10      gset_output(io);
11
12      // Pull low for >480uS (master reset pulse)
13      gset_output_low(io);
14      _delay_us(480);
15
16      // Configure for input
17      gset_input_hiz(io);
18      _delay_us(70);
19
20      // Look for the line pulled low by a slave
21      uint8_t result = gread_bit(io);
22
23      // Wait for the presence pulse to finish
24      // This should be less than 240uS, but the master is expected to stay
25      // in Rx mode for a minimum of 480uS in total
26      _delay_us(460);
27
```

```
    return result == 0;
}

/**
 * Output a Write-0 or Write-1 slot on the One Wire bus
 * A Write-1 slot is generated unless the passed value is zero
 */
static void onewire_write_bit(const gpin_t* io, uint8_t bit)
{
    if (bit != 0) { // Write high

        // Pull low for less than 15uS to write a high
        gset_output_low(io);
        _delay_us(5);
        gset_output_high(io);

        // Wait for the rest of the minimum slot time
        _delay_us(55);

    } else { // Write low

        // Pull low for 60 - 120uS to write a low
        gset_output_low(io);
        _delay_us(55);

        // Stop pulling down line
        gset_output_high(io);

        // Recovery time between slots
        _delay_us(5);
    }
}

// One Wire timing is based on this Maxim application note
// https://www.maximintegrated.com/en/app-notes/index.mvp/id/126
void onewire_write(const gpin_t* io, uint8_t byte)
{
    // Configure for output
    gset_output_high(io);
    gset_output(io);

    for (uint8_t i = 8; i != 0; --i) {

        onewire_write_bit(io, byte & 0x1);

        // Next bit (LSB first)
        byte >>= 1;
    }
}
```

```
77
78  /**
79   * Generate a read slot on the One Wire bus and return the bit value
80   * Return 0x0 or 0x1
81   */
82  static uint8_t onewire_read_bit(const gpin_t* io)
83  {
84      // Pull the 1-wire bus low for >1uS to generate a read slot
85      gset_output_low(io);
86      gset_output(io);
87      _delay_us(1);
88
89      // Configure for reading (releases the line)
90      gset_input_hiz(io);
91
92      // Wait for value to stabilise (bit must be read within 15uS of read slot)
93      _delay_us(10);
94
95      uint8_t result = gread_bit(io) != 0;
96
97      // Wait for the end of the read slot
98      _delay_us(50);
99
100     return result;
101 }
102
103 uint8_t onewire_read(const gpin_t* io)
104 {
105     uint8_t buffer = 0x0;
106
107     // Configure for input
108     gset_input_hiz(io);
109
110     // Read 8 bits (LSB first)
111     for (uint8_t bit = 0x01; bit; bit <<= 1) {
112
113         // Copy read bit to least significant bit of buffer
114         if (onewire_read_bit(io)) {
115             buffer |= bit;
116         }
117     }
118
119     return buffer;
120 }
121
122 void onewire_match_rom(const gpin_t* io, uint8_t* address)
123 {
124     // Write Match Rom command on bus
125     onewire_write(io, 0x55);
```

```
126
127     // Send the passed address
128     for (uint8_t i = 0; i < 8; ++i) {
129         onewire_write(io, address[i]);
130     }
131 }
132
133 void onewire_skiprom(const gpin_t* io)
134 {
135     onewire_write(io, 0xCC);
136 }
137
138 /**
139  * Search procedure for the next ROM addresses
140  *
141  * This algorithm is bit difficult to understand from the diagrams in Maxim's
142  * datasheets and app notes, though its reasonably straight forward once
143  * understood.  I've used the name "last zero branch" instead of Maxim's name
144  * "last discrepancy", since it describes how this variable is used.
145  *
146  * A device address has 64 bits. With multiple devices on the bus, some bits
147  * are ambiguous.  Each time an ambiguous bit is encountered, a zero is written
148  * and the position is marked.  In subsequent searches at ambiguous bits, a one
149  * is written at this mark, zeros are written after the mark, and the bit in
150  * the previous address is copied before the mark. This effectively steps
151  * through all addresses present on the bus.
152  *
153  * For reference, see either of these documents:
154  *
155  *  - Maxim application note 187: 1-Wire Search Algorithm
156  *    https://www.maximintegrated.com/en/app-notes/index.mvp/id/187
157  *
158  *  - Maxim application note 937: Book of iButton® Standards (pages 51-54)
159  *    https://www.maximintegrated.com/en/app-notes/index.mvp/id/937
160  *
161  * @see onewire_search()
162  * @returns true if a new address was found
163  */
164 static bool _search_next(const gpin_t* io, onewire_search_state* state)
165 {
166     // States of ROM search reads
167     enum {
168         kConflict = 0b00,
169         kZero = 0b10,
170         kOne = 0b01,
171     };
172
173     // Value to write to the current position
174     uint8_t bitValue = 0;
```

```
175
176     // Keep track of the last zero branch within this search
177     // If this value is not updated, the search is complete
178     int8_t localLastZeroBranch = -1;
179
180     for (int8_t bitPosition = 0; bitPosition < 64; ++bitPosition) {
181
182         // Calculate bitPosition as an index in the address array
183         // This is written as-is for readability. Compilers should reduce this to bit shifts and t
184         uint8_t byteIndex = bitPosition / 8;
185         uint8_t bitIndex = bitPosition % 8;
186
187         // Configure bus pin for reading
188         gset_input_hiz(io);
189
190         // Read the current bit and its complement from the bus
191         uint8_t reading = 0;
192         reading |= onewire_read_bit(io); // Bit
193         reading |= onewire_read_bit(io) << 1; // Complement of bit (negated)
194
195         switch (reading) {
196             case kZero:
197             case kOne:
198                 // Bit was the same on all responding devices: it is a known value
199                 // The first bit is the value we want to write (rather than its complement)
200                 bitValue = (reading & 0x1);
201                 break;
202
203             case kConflict:
204                 // Both 0 and 1 were written to the bus
205                 // Use the search state to continue walking through devices
206                 if (bitPosition == state->lastZeroBranch) {
207                     // Current bit is the last position the previous search chose a zero: send one
208                     bitValue = 1;
209
210                 } else if (bitPosition < state->lastZeroBranch) {
211                     // Before the lastZeroBranch position, repeat the same choices as the previous
212                     bitValue = state->address[byteIndex] & (1 << bitIndex);
213
214                 } else {
215                     // Current bit is past the lastZeroBranch in the previous search: send zero
216                     bitValue = 0;
217                 }
218
219                 // Remember the last branch where a zero was written for the next search
220                 if (bitValue == 0) {
221                     localLastZeroBranch = bitPosition;
222                 }
223
```

```
                break;

            default:
                // If we see "11" there was a problem on the bus (no devices pulled it low)
                return false;
        }

        // Write bit into address
        if (bitValue == 0) {
            state->address[byteIndex] &= ~(1 << bitIndex);
        } else {
            state->address[byteIndex] |= (bitValue << bitIndex);
        }

        // Configure for output
        gset_output_high(io);
        gset_output(io);

        // Write bit to the bus to continue the search
        onewire_write_bit(io, bitValue);
    }

    // If the no branch points were found, mark the search as done.
    // Otherwise, mark the last zero branch we found for the next search
    if (localLastZeroBranch == -1) {
        state->done = true;
    } else {
        state->lastZeroBranch = localLastZeroBranch;
    }

    // Read a whole address - return success
    return true;
}

static inline bool _search_devices(uint8_t command, const gpin_t* io, onewire_search_state* state)
{
    // Bail out if the previous search was the end
    if (state->done) {
        return false;
    }

    if (!onewire_reset(io)) {
        // No devices present on the bus
        return false;
    }

    onewire_write(io, command);
    return _search_next(io, state);
}
```

```
273
274 bool onewire_search(const gpin_t* io, onewire_search_state* state)
275 {
276     // Search with "Search ROM" command
277     return _search_devices(0xF0, io, state);
278 }
279
280 bool onewire_alarm_search(const gpin_t* io, onewire_search_state* state)
281 {
282     // Search with "Alarm Search" command
283     return _search_devices(0xEC, io, state);
284 }
285
286 bool onewire_check_rom_crc(onewire_search_state* state)
287 {
288     // Validate bits 0..56 (bytes 0 - 6) against the CRC in byte 7 (bits 57..63)
289     return state->address[7] == crc8(state->address, 7);
290 }
```

onewire.h

```
1  #pragma once
2
3  #include "pindef.h"
4
5  // C
6  #include <stdbool.h>
7  #include <stdint.h>
8  #include <string.h>
9
10 /**
11  * State for the onewire_search function
12  * This must be initialised with onewire_search_init() before use.
13  */
14 typedef struct onewire_search_state {
15
16     // The highest bit position where a bit was ambiguous and a zero was written
17     int8_t lastZeroBranch;
18
19     // Internal flag to indicate if the search is complete
20     // This flag is set once there are no more branches to search
21     bool done;
22
23     // Discovered 64-bit device address (LSB first)
24     // After a successful search, this contains the found device address.
25     // During a search this is overwritten LSB-first with a new address.
26     uint8_t address[8];
27
```

```
28  } onewire_search_state;
29
30  /**
31   * Send a reset pulse
32   *
33   * Returns true if One Wire devices were detected on the bus
34   */
35  bool onewire_reset(const gpin_t* io);
36
37  /**
38   * Write a byte as described in Maxim's One Wire protocol
39   */
40  void onewire_write(const gpin_t* io, uint8_t byte);
41
42  /**
43   * Read a byte as described in Maxim's One Wire protocol
44   */
45  uint8_t onewire_read(const gpin_t* io);
46
47  /**
48   * Skip sending a device address
49   */
50  void onewire_skiprom(const gpin_t* io);
51
52  /**
53   * Address a specific device
54   */
55  void onewire_match_rom(const gpin_t* io, uint8_t* address);
56
57  /**
58   * Reset a search state for use in a search
59   */
60  inline void onewire_search_init(onewire_search_state* state)
61  {
62      state->lastZeroBranch = -1;
63      state->done = false;
64
65      // Zero-fill the address
66      memset(state->address, 0, sizeof(state->address));
67  }
68
69  /**
70   * Look for the next slave address on the bus
71   *
72   * Before the first search call, the state parameter must be initialised using
73   * onewire_init_search(state). The same state must be passed to subsequent calls
74   * to discover all available devices.
75   *
76   * The caller is responsible for performing a CRC check on the result if desired.
```

```
 *
 * @returns true if a new address was found
 */
bool onewire_search(const gpin_t* io, onewire_search_state* state);

/**
 * Look for the next slave address on the bus with an alarm condition
 * @see onewire_search()
 */
bool onewire_alarm_search(const gpin_t* io, onewire_search_state* state);

/**
 * Return true if the CRC byte in a ROM address validates
 */
bool onewire_check_rom_crc(onewire_search_state* state);
```

pindef.c

```
#include "pindef.h"

void gset_input_pullup(const gpin_t* pin) {
    *(pin->ddr) &= ~_BV(pin->bit);
    gset_output_high(pin);
}

void gset_input_hiz(const gpin_t* pin) {
    *(pin->ddr) &= ~_BV(pin->bit);
    gset_output_low(pin);
}

void gset_output(const gpin_t* pin) {
    *(pin->ddr) |= _BV(pin->bit);
}

void gset_output_high(const gpin_t* pin) {
    *(pin->port) |= _BV(pin->bit);
}

void gset_output_low(const gpin_t* pin) {
    *(pin->port) &= ~_BV(pin->bit);
}

void gset_bit(const gpin_t* pin) {
    *(pin->port) |= _BV(pin->bit);
}

void gclear_bit(const gpin_t* pin) {
    *(pin->port) &= ~_BV(pin->bit);
}
```

```
32
33  uint8_t gread_bit(const gpin_t* pin) {
34      return *(pin->pin) & _BV(pin->bit);
35  }
```

pindef.h

```
1   #pragma once
2
3   // AVR
4   #include <avr/io.h>
5
6   // C
7   #include <stdint.h>
8
9   /**
10   * Generic AVR port pin
11   */
12  typedef struct gpin_t {
13      // Pointers to PORT and PIN and DDR registers
14      volatile uint8_t *port;
15      volatile uint8_t *pin;
16      volatile uint8_t *ddr;
17
18      // Bit number in PORT
19      uint8_t bit;
20  } gpin_t;
21
22  void gset_input_pullup(const gpin_t* pin);
23  void gset_input_hiz(const gpin_t* pin);
24  void gset_output(const gpin_t* pin);
25  void gset_output_high(const gpin_t* pin);
26  void gset_output_low(const gpin_t* pin);
27  void gset_bit(const gpin_t* pin);
28  void gclear_bit(const gpin_t* pin);
29  uint8_t gread_bit(const gpin_t* pin);
```