What is One Wire Serial Communication

- Serial data transmission over one wire(duh)
- Great for cheap applications that don't require high speed data transfer
- In theory, all AVR processors can handle One Wire Communication
- Timing is extremely important
    - The only way to discern signals from other signals

How Does it Work?

- Two basic implementations
  - Polled Implementations(software only)
  - Interrupt driven (counter required)
- Dallas 1-Wire Protocol
  - Designed by the Dallas Semiconductor Corporation
  - Can use either interrupt driven or polled implementations
  - Polled requires no external hardware
  - Half Duplex(transmitting or receiving not Both simultaneously)

Dallas 1-Wire

- Signal is idle High, need pull up resistor
- Signaling is divided into time slots of 60µs
- Master initiates every communication
  - Regardless of direction
- 5 basic signals: (Write 1, Write 0, Read, Reset, Presence)
- LSB sent first

Overall Communication process

All 1-Wire devices follow this basic sequence:

1. The Master sends the Reset pulse.
2. The Slave(s) respond with a Presence pulse.
3. Master sends a ROM command
4. Master sends a Memory command

How To Send Bits

- In order to send bits in OW protocol it requires precise timing and precise waveforms
- Errors in these waveforms, or the timing can deliver corrupt data and requires digital error checking

Bus Signals

- Write 1
  - The Master pulls the bus low for 15µs then releases for remainder of 60µs window

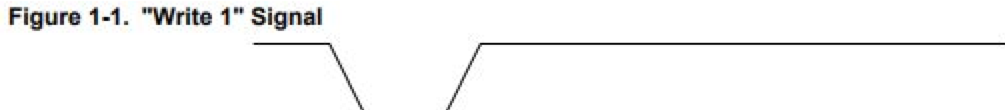


Figure 1-1. "Write 1" Signal

Bus Signals Continued

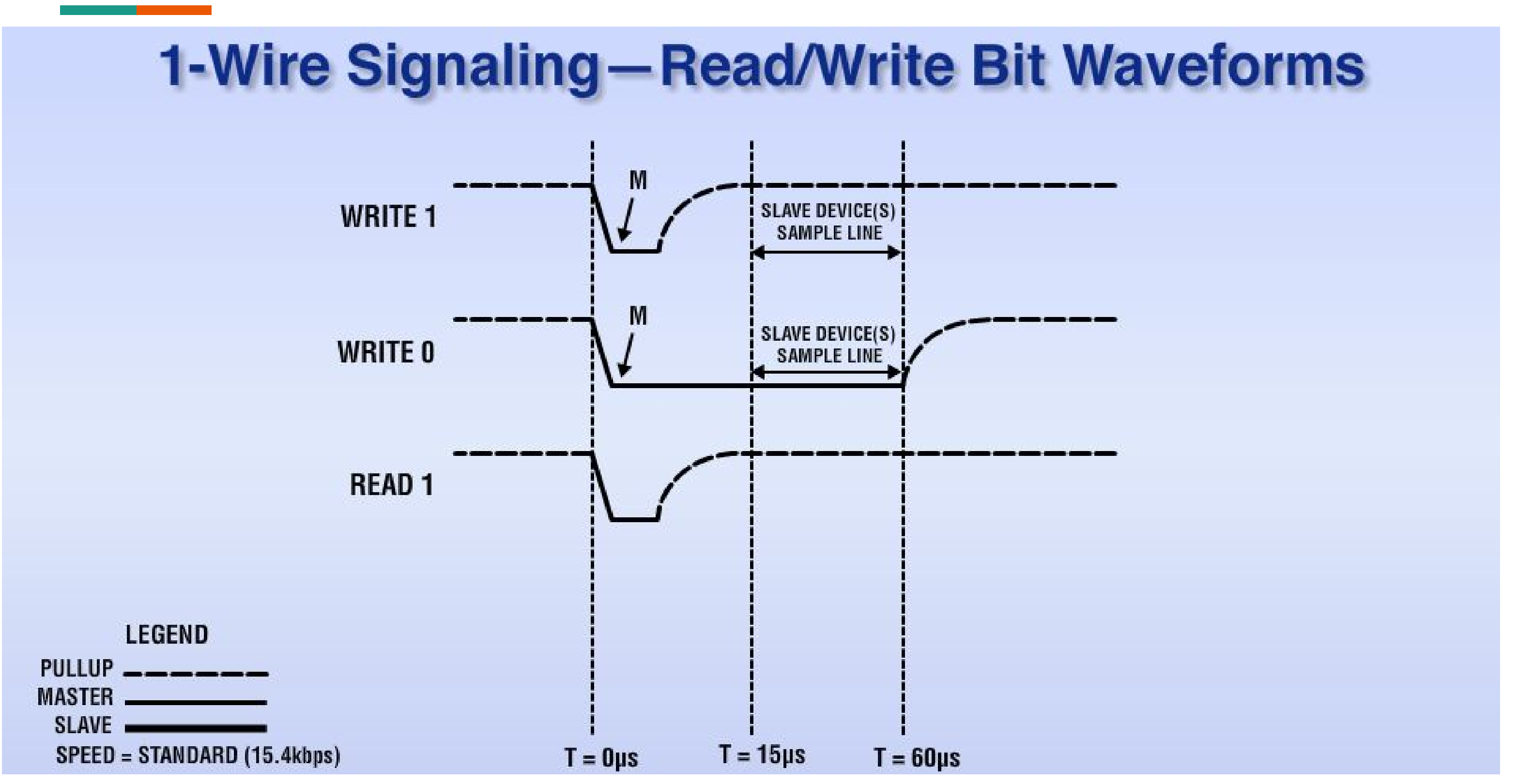
- Write 0
  - The master pulls the bus low for 60µs

Figure 1-2. "Write 0" Signal

- Read
  - Master pulls bus low for 1 to 15µs
  - The slave then releases the line if it wants to write 1 or keeps the line low for write 0 for remainder of 60 us window.

Figure 1-3. "Read" Signal

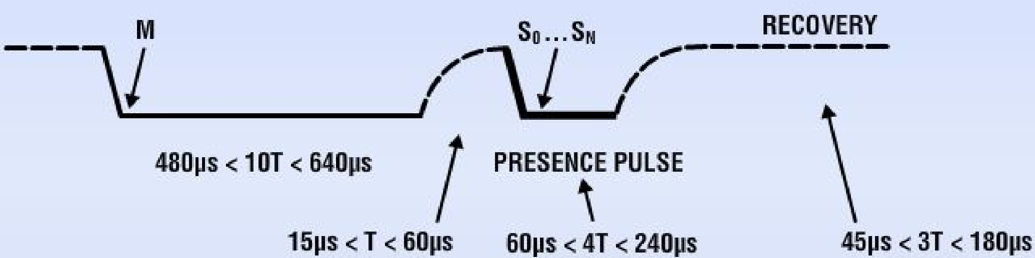
Timing Diagram of OW Signals



Reset/Presence Signal

- Reset Signal
  - The master pulls the line low for 480µs and then releases it
  - If slave is present then the presence signal should be seen
  - If no presence signal is detected there is no device connected
- Presence Signal
  - After the Reset Signal has been sent it is the slave that must pull the line low for 60µs
  - Must pull the line low within 60µs

Reset Presence Signal Waveform



Generating Signals through Software

```
#define PINNUMBER 2
#define A 6
#define B 64
#define C 60
#define D 10
#define E 9
#define F 55
#define G 0
#define H 480
#define I 70
#define J 410
```

- Define delay values
- Define our BUS pin

- Initialize the bus as an output

```
DDRF |= (1<<PF5);
```

Structure of delayMicroseconds function

```
void delayMicroseconds(uint16_t value){// 16 cycles

  if(value <= 2)return; // 2 cycles

  value <<= 1; // 2 cycles
// We must subtract 20 cycles from our iteration loop in order to get exact timing
// The above lines contribute to our timing so we must subtract that from our loop number
  value -= 5;  // 2 cycles
// Every iteration of the loop is 0.5 microseconds so we double our value in order to get exact microsecond values
  __asm__ __volatile__(
    "1: sbiw %0,1" "\n\t" //2 cycles
    "brne 1b" : "=w"(value):"0" (value)// 2 cycles
    );
}
```

One Wire Write function

```
void OWWrite(uint8_t bit){
  if(bit){
    //Write '1' bit
    digitalWrite(PINNUMBER,0x00);//Drives bus low
    delayMicroseconds(A);
    digitalWrite(PINNUMBER,0x00);//Release the bus
    delayMicroseconds(D);
  }
  else{
    //Write '0' bit
    digitalWrite(PINNUMBER,0x00);//Drives bus low
    delayMicroseconds(C);
    digitalWrite(PINNUMBER,0x01);//Releases the bus
    delayMicroseconds(D);
  }
}
```

C++ One Wire Write function

```
void OWWriteBit(uint8_t bit){

  if(bit == 1){
  PORTF &= ~(1<<PF5);
  delayMicroseconds(A);
  PORTF |= (1<<PF5);
  delayMicroseconds(B);
  }

  else{
  PORTF &= ~(1<<PF5);
  delayMicroseconds(C);
  PORTF |= (1<<PF5);
  delayMicroseconds(D);
  }

}
```

One Wire Read Function

```
uint8_t OWRead(void){
  uint8_t result;

  digitalWrite(PINNUMBER,0x00);//Drives bus low
  delayMicroseconds(A);
  digitalWrite(PINNUMBER,0x01);//Releases the bus
  delayMicroseconds(E);
  pinMode(PINNUMBER,INPUT);//Makes the pin an input so we can read from the slave
  result = digitalRead(PINNUMBER)& 0x01// Sample the bit value from the slave
  delayMicroseconds(F);
  pinMode(PINNUMBER,OUTPUT);//Makes the pin an output so we can write for the next function call

  return result;
}
```

C++ One Wire Read Bit Function

```
uint8_t OWReadBit(void){
  uint8_t result = 0;

  PORTF &= ~(1<<PF5);
  delayMicroseconds(10);
  PORTF |= (1<<PF5);
  delayMicroseconds(20);
  if(PINF & (1<<PF5)){
    result = HIGH;
  }
  delayMicroseconds(30);
  return result;

}
```

One Wire Reset/Presence Signal

```
uint8_t OWResetPresence(void){
uint8_t result;

delayMicroseconds(G);
digitalWrite(PINNUMBER,0x00);//Drives bus low
delayMicroseconds(H);
digitalWrite(PINNUMBER,0x00);//Releases the bus
delayMicroseconds(I);
pinMode(PINNUMBER,INPUT);
result = digitalRead(PINNUMBER)^0x01//Sample presence pulse from slave
delayMicroseconds(J);
pinMode(PINNUMBER,OUTPUT);
return result;// Return sample presence pulse reset


}
```

C++ One Wire Reset/Presence Signal

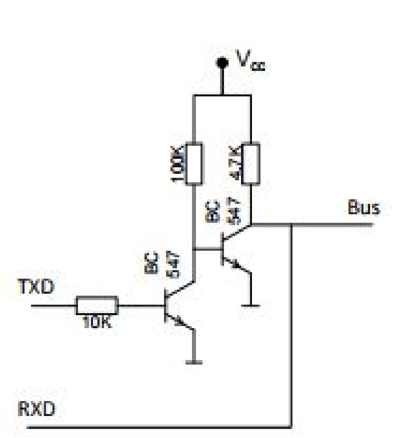
```
uint8_t OWResetPresence(void){

uint8_t result = LOW;
  PORTF &= ~(1<<PF5);
  delayMicroseconds(H);
  PORTF |= (1<<PF5);
  delayMicroseconds(55);
  if(PINF&(1<<PF5)){
    result = HIGH;
  }
  return result;

}
```

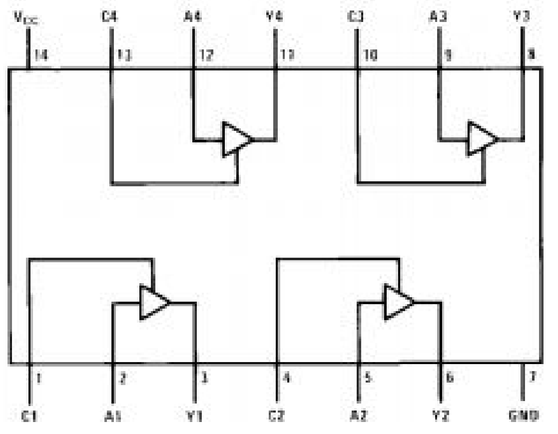
Generating the Signals

- Software Only
  - Simply changing the direction and value of a GPIO and generating required delay is sufficient
- With UART
  - Requires both TXD and RXD pins to be connected to the bus
  - Need tri-state or open-collector buffer so the slave can pull the line low.

IC for UART Signals

DM74LS126A Tri-state Buffer IC





Function Table

Y = A

Inputs		Output
A	C	Y
L	H	L
H	H	H
X	L	Hi-Z

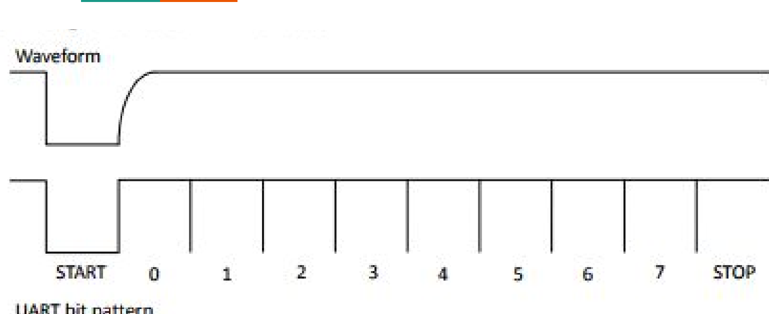
H = HIGH Logic Level
L = LOW Logic Level
X = Either LOW or HIGH Logic Level
Hi-Z = 3-STATE (Outputs are disabled)

Generating UART Signals

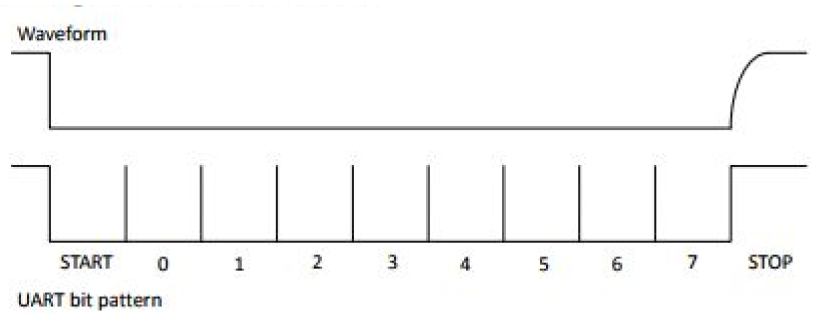
- Baud rate is equivalent to bits/sec
- In order to transmit 1-wire framed data from UART platform we must frame the data into the same time slots as in the polled implementation
- One UART data frame is used to generate one waveform bit of 1 wire data or one RESET/Presence signal

Signal	Baud rate	Transmit value	Receive value
Write 1	115200	FFh	FFh
Write 0	115200	00h	00h
Read	115200	FFh	FFh equals a '1' bit Anything else equals a '0' bit
Reset/Presence	9600	F0h	F0h equals no presence. Anything else equals presence.

Framing 1-Wire from UART

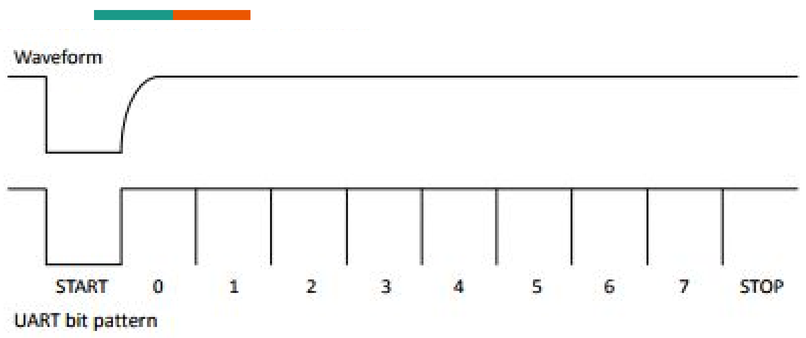


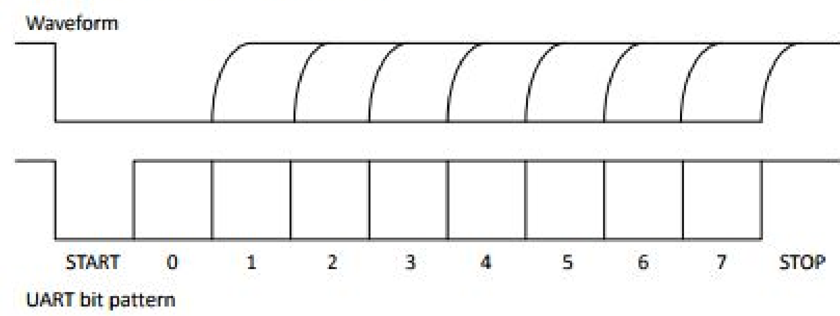
- To match waveform UART data frame must match 1-Wire data frame
- All UART signals initiate with a start bit “0” or LOW
- To “Write 1” in one wire format all bits in UART Frame after Start bit must be HIGH or “1”



- To “Write 0” all bits in UART data frame are “0” or LOW
- Stop bit is always HIGH or “1”

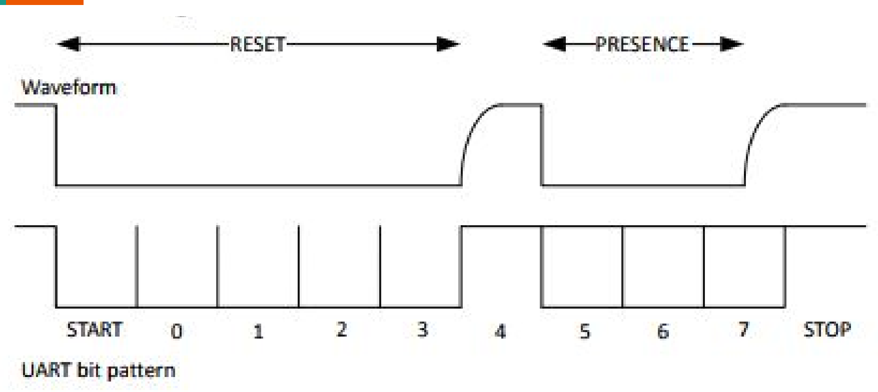
Generating One Wire Frame From UART





- “Read 1” Start bit is LOW and slave writes bit “1” or HIGH in the remaining data bits in UART frame
- “Read 0” start bit is LOW and the slave drives the bus low by writing “0” to the rest of the UART bit pattern and drives the bus HIGH for the STOP bit.

Reset/Presence Signal with UART



- UART bits 0-3 are LOW, 4 is HIGH, and Slave writes UART bits 5,6,&7 LOW
- Stop bit for this signal is HIGH

Overall Communication process

All 1-Wire devices follow this basic sequence(MUST FOLLOW):

1. The Master sends the Reset pulse.
2. The Slave(s) respond with a Presence pulse.
3. Master sends a ROM command
4. Master sends a Memory command

ROM Commands

- All 1-Wire devices contain a 64 bit identifier stored in ROM
- ROM Commands address those 64 bit identifiers
- After bus master detects a presence pulse it can issue a ROM command to detect how many slaves are on the bus, and which slave to address.

ROM Commands MAX31820

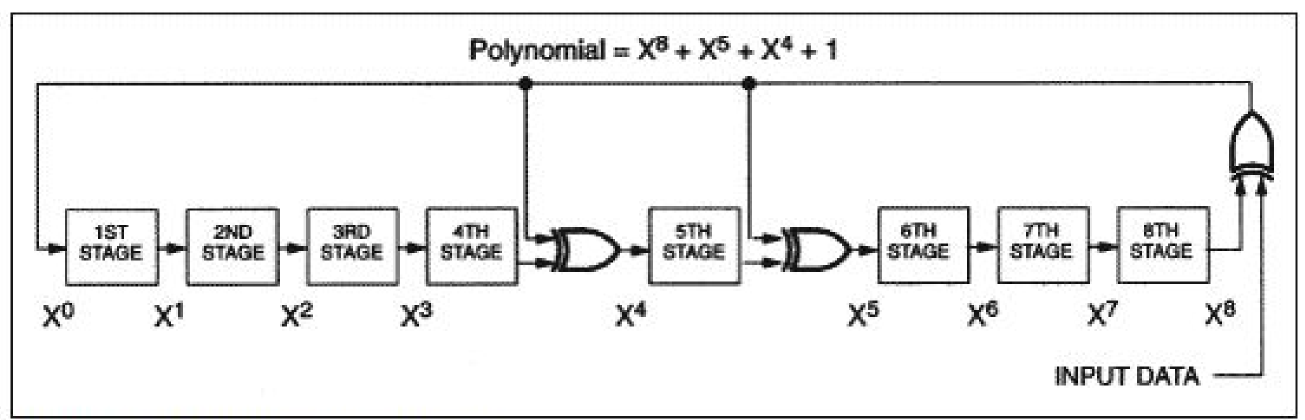
- Read ROM(33h)-Reads the ROM code of single slave device. If there are multiple slave devices and the command is issued, data collision will occur.
- Skip ROM(CCh)-Sends data for addressing to all connected slave devices
- Match ROM(55h)-Is used to address individual slave devices on the bus
- Search ROM(F0h)-Is used to get the ID of slave devices if not known

Cyclic Redundancy Check

- CRC is used an every ROM command to check for data integrity.
- CRC - is an error detection code for digital networks.
- It sends a check value along with the data that is transmitted. This value is determined by dividing the data by a nominal value and returning the remainder.
- If once the data is received and the remainder value is not the same, after the received data has been divided by the nominal value, then we know the data has been corrupted.

How CRC works

- We can think of CRC as an 8 bit shift register with feedback



How CRC Works

- At the beginning of a 64 bit Read ROM command the CRC shift register has an initial value of zero. CRC = 0x00
- The LSB of the 64 bit identifier is XORd with the LSB of the input data. In the case of 64 bit identifiers the LSB is a family CRC code(ex: 0xA2). So after the first byte of data has been processed the CRC register will contain the value of the CRC code. CRC = 0xA2
- After the entire 64 bit identifier has been shifted into and out of the CRC register the register should have a final value of 0x00 if no data was corrupted. If data was corrupted we would have a value inside the CRC shift register and the data would have to be retransmitted.

CRC Computation of 1-Wire(Look up Table)

1. Dallas Semiconductor Corp. provides a table for CRC calculations. This table rapidly increase CRC calculations and can process entire bytes of data per loop run.
2. CRC register is initially zero we then shift in the first byte of 64 bit ID into register
3. Then we XOR the next byte of data with the CRC register
4. Using pgm_read_byte we point to the index of our table and set our CRC register to that new value. We are indexing by the XORd value of the CRC register and the input data.
5. We repeat steps 4 and 5 until we are done with all 64 bits or 8 bytes of our ID.
6. If CRC is 0 then no data was corrupted else data has been corrupted.

CRC Function Code Example

```
static const uint8_t PROGMEM dscrc_table[] = {
      0, 94,188,226, 97, 63,221,131,194,156,126, 32,163,253, 31, 65,
    157,195, 33,127,252,162, 64, 30, 95,  1,227,189, 62, 96,130,220,
     35,125,159,193, 66, 28,254,160,225,191, 93,  3,128,222, 60, 98,
    190,224,  2, 92,223,129, 99, 61,124, 34,192,158, 29, 67,161,255,
     70, 24,250,164, 39,121,155,197,132,218, 56,102,229,187, 89,  7,
    219,133,103, 57,186,228,  6, 88, 25, 71,165,251,120, 38,196,154,
    101, 59,217,135,  4, 90,184,230,167,249, 27, 69,198,152,122, 36,
    248,166, 68, 26,153,199, 37,123, 58,100,134,216, 91,  5,231,185,
    140,210, 48,110,237,179, 81, 15, 78, 16,242,172, 47,113,147,205,
     17, 79,173,243,112, 46,204,146,211,141,111, 49,178,236, 14, 80,
    175,241, 19, 77,206,144,114, 44,109, 51,209,143, 12, 82,176,238,
     50,108,142,208, 83, 13,239,177,240,174, 76, 18,145,207, 45,115,
    202,148,118, 40,171,245, 23, 73,  8, 86,180,234,105, 55,213,139,
     87,  9,235,181, 54,104,138,212,149,203, 41,119,244,170, 72, 22,
    233,183, 85, 11,136,214, 52,106, 43,117,151,201, 74, 20,246,168,
    116, 42,200,150, 21, 75,169,247,182,232, 10, 84,215,137,107, 53};
```

```
uint8_t OneWire::crc8(const uint8_t *addr, uint8_t len)
{
        uint8_t crc = 0;

        while (len--) {
                crc = pgm_read_byte(dscrc_table + (crc ^ *addr++));
        }
        return crc;
}
```

CRC Look Up Method

Table 1. Table Lookup Method for Computing 1-Wire CRC

Current CRC Value (= Current Table Index)	Input Data	New Index (= Current CRC xor Input Data)	Table (New Index) (= New CRC Value)
0000 0000 = 00 hex	0000 0010 = 02 hex	(00 H xor 02 H) = 02 hex = 2 dec	Table[2]= 1011 1100 = BC hex = 188 dec
1011 1100 = BC hex	0001 1100 = 1C hex	(BC H xor 1C H) = A0 hex = 160 dec	Table[160]= 1010 1111 = AF hex = 175 dec
1010 1111 = AF hex	1011 1000 = B8 hex	(AF H xor B8 H) = 17 hex = 23 dec	Table[23]= 0001 1110 = 1E hex = 30 dec
0001 1110 = 1E hex	0000 0001 = 01 hex	(1E H xor 01 H) = 1 F hex = 31 dec	Table[31]= 1101 110 = DC hex = 220 dec
1101 1100 = DC hex	0000 0000 = 00 hex	(DC H xor 00 H) = DC hex = 220 dec	Table[220]= 1111 0100 = F4 hex = 244 dec
11110100 = F4 hex	0000 0000 = 00 hex	(F4 H xor 00 H) = F4 hex = 244 dec	Table [244]= 0001 0101 = 15 hex = 21 dec
0001 0101 = 15 hex	0000 0000 = 00 hex	(15 H xor 00 H) = 15 hex = 21 dec	Table[21]= 1010 0010 = A2 hex = 162 dec
1010 0010 = A2 hex	10100010 = A2 hex	(A2 H xor A2 H) = hex = 0 dec	Table[0]=0000 0000 = 00 hex = 0 dec

This table shows how each byte of data gets shifted into our CRC register and how we calculate the next index for our CRC value.

Overall Communication process

All 1-Wire devices follow this basic sequence(MUST FOLLOW):

1. The Master sends the Reset pulse.
2. The Slave(s) respond with a Presence pulse.
3. Master sends a ROM command
4. Master sends a Memory command

Memory Commands

- Are commands specific to slave devices or a class of devices
- They deal with the internal memory and registers of slave devices
- Therefore for each device memory commands are specific to that device

MAX31820 Memory Registers

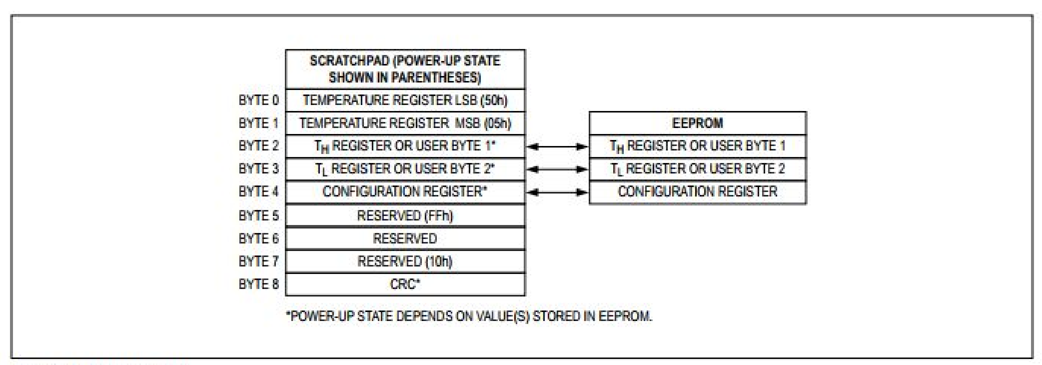


Figure 4. Memory Map

MAX31820 - Ambient Temperature Sensor

Configuration Register Format

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0	R1	R0	1	1	1	1	1

- Memory commands specific to this device would address this device's internal memory registers
- TH and TL would address BYTE 2 and BYTE 3 of the temperature sensors internal scratchpad memory

MAX31820 Configuration Registers

R1	R0	RESOLUTION (BITS)	MAX CONVERSION TIME	
0	0	9	93.75ms	($t_{CONV}/8$)
0	1	10	187.5ms	($t_{CONV}/4$)
1	0	11	375ms	($t_{CONV}/2$)
1	1	12	750ms	(t_{CONV})

- Bits R1 and R0 control the resolution of the temperature reading
- Better resolution slower conversion time, and faster conversion time less resolution

MAX31820 Function Command Set

Table 3. MAX31820 Function Command Set

COMMAND	DESCRIPTION	PROTOCOL	1-Wire BUS ACTIVITY AFTER COMMAND IS ISSUED
Convert T (Note 1)	Initiates temperature conversion.	44h	The device transmits conversion status to master (not applicable for parasite-powered devices).
Read Scratchpad (Note 2)	Reads the entire scratchpad including the CRC byte.	BEh	The device transmits up to 9 data bytes to master.
Write Scratchpad (Note 3)	Writes to scratchpad bytes 2, 3, and 4 (T_H, T_L, and configuration registers).	4Eh	The master transmits 3 data bytes to the device.
Copy Scratchpad (Note 1)	Copies T_H, T_L, and configuration register data from the scratchpad to EEPROM.	48h	None.
Recall E^2	Recalls T_H, T_L, and configuration register data from EEPROM to the scratchpad.	B8h	The device transmits recall status to the master.
Read Power Supply	Signals the device's power-supply mode to the master.	B4h	The device transmits supply status to the master.

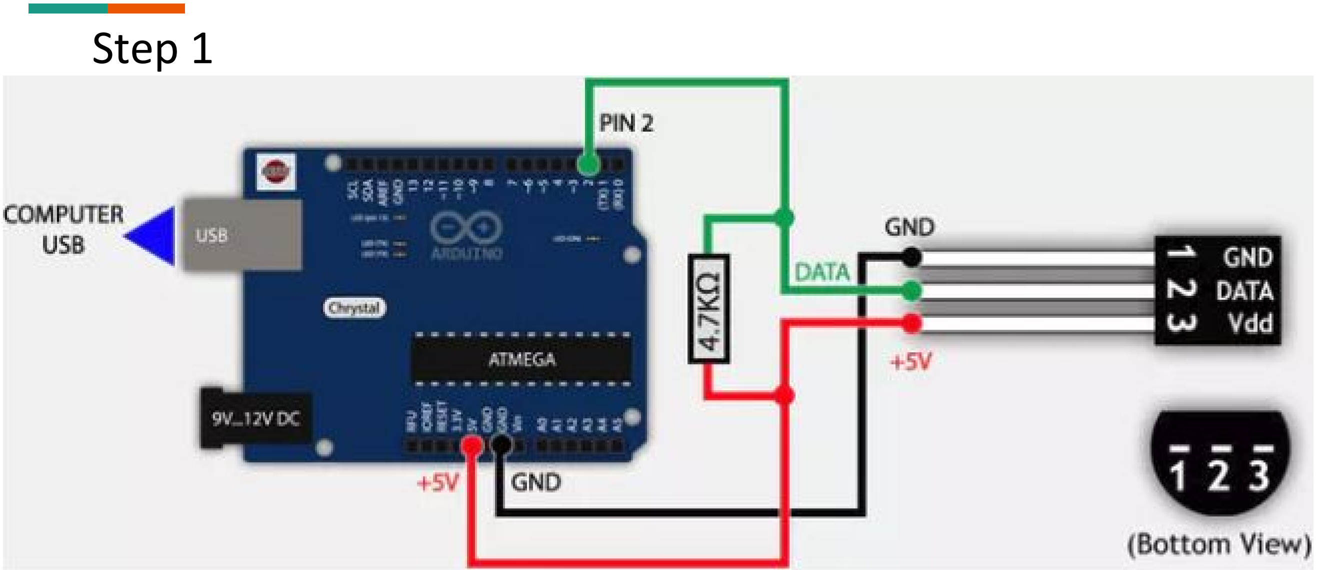
Overall Communication process

All 1-Wire devices follow this basic sequence:

1. The Master sends the Reset pulse.
2. The Slave(s) respond with a Presence pulse.
3. Master sends a ROM command
4. Master sends a Memory command

1 Wire Digital Temperature Sensor Implementation

Step 1



- Assemble this simple circuit, the resistor is needed for the idle state of the bus(HIGH)
- Also because it will act as a pull up resistor

1 Wire Digital Temperature Sensor Implementation

Step 2

Download Necessary Libraries

- Download OneWire Arduino Library
- Download DallasTemperature Arduino Library
- Place both libraries into your Arduino library path
- Include both libraries in your C++ code

1 Wire Digital Temperature Sensor Implementation

Step 3 Arduino Code

```
/********************************************************************/
// First we include the libraries
#include <OneWire.h>
#include <DallasTemperature.h>
/********************************************************************/
// Data wire is plugged into pin 2 on the Arduino
#define ONE_WIRE_BUS 2
/********************************************************************/
// Setup a oneWire instance to communicate with any OneWire devices
// (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);
/********************************************************************/
// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);
/********************************************************************/
void setup(void)
{
 // start serial port
 Serial.begin(9600);
 Serial.println("Dallas Temperature IC Control Library Demo");
 // Start up the library
 sensors.begin();
}
void loop(void)
{
 // call sensors.requestTemperatures() to issue a global temperature
 // request to all devices on the bus
/********************************************************************/
 Serial.print(" Requesting temperatures...");
 sensors.requestTemperatures(); // Send the command to get temperature readings
 Serial.println("DONE");
/********************************************************************/
 Serial.print("Temperature is: ");
 Serial.print(sensors.getTempCByIndex(0)); // Why "byIndex"?
   // You can have more than one DS18B20 on the same bus.
   // 0 refers to the first IC on the wire
   delay(1000);
}
```

OneWire and DallasTemperature functions

- OneWire oneWire(ONE_WIRE_BUS) - this is a data structure defined in the one wire library
- DallasTemperature sensors(&oneWire) - this is another data structure defined in the DallasTemperature library that is an array of 8 bytes.(that is why the structure must be passed as a pointer)

OneWire and DallasTemperature functions

- sensors.begin - is a subroutine in the DallasTemperature library that identifies the type of temperature sensor on the bus and initializes the type of power supply the temperature sensors requires.
- sensors.requestTemperatures - sends a reset/presence signal to begin communication with device. Once presence is detected skip ROM command is sent because only one temperature sensor is on the bus. Then we issue a memory command that begins to send data of the temperature back to the master device.

OneWire and DallasTemperature functions

- sensors.getTempCByIndex(0) - This function returns the temperature in celsius detected by the temperature sensor. If device is disconnected during process it will return “DEVICE_DISCONNECTED_C”.

References

https://en.wikipedia.org/wiki/1-Wire

http://www.atmel.com/images/Atmel-2579-Dallas-1Wire-Master-on-tinyAVR-and-megaAVR_ApplicationNote_AVR318.pdf

https://www.maximintegrated.com/en/app-notes/index.mvp/id/126

https://forum.pjrc.com/threads/23939-Strange-behavior-on-the-Onewireslave-library?p=33608&viewfull=1#post33608

https://create.arduino.cc/projecthub/TheGadgetBoy/ds18b20-digital-temperature-sensor-and-arduino-9cc806

https://www.hacktronics.com/Tutorials/arduino-1-wire-address-finder.html

https://en.wikipedia.org/wiki/Cyclic_redundancy_check

https://www.maximintegrated.com/en/app-notes/index.mvp/id/27

https://www.maximintegrated.com/en/products/1-wire/flash/overview/

Questions

1. How long are the time divisions of each bit level command?
2. What is the Idle state of the bus line?
3. What are the types of implementations of one wire?
4. Which implementation requires external hardware?
5. How does the master know there is a slave device connected?
6. Where are the 64 bit IDs of connected devices stored?
7. Which signal takes more than 1 time slot to transmit?
8. What device is needed for one wire UART?
9. What is the waveform of the write 1 signal?
10. Can data be both transmitted and received simultaneously by the one wire protocol?

Answers

1. 60 microseconds
2. HIGH
3. Polled, UART(interrupt driven)
4. UART
5. Presence signal
6. ROM
7. Reset/Presence Signal
8. Tri-state buffer
9. Draw
10. No