AVR 8-bit Microcontrollers

AVR318: Dallas 1-Wire Master on tinyAVR and megaAVR

APPLICATION NOTE

Introduction

Dallas 1-Wire® devices are unique in that only one wire in addition to ground is needed to communicate with a device. Power supply and communications are handled through only one connection. To communicate with a Dallas 1-Wire device, only one general purpose I/O pin is needed. This application note shows how a 1-Wire master can be implemented on an Atmel® AVR®, either in software only, or utilizing the U(S)ART module.

Features

- Supports standard speed Dallas 1-Wire protocol
- Compatible with all AVRs
- Polled or interrupt-driven implementation
- Polled implementation requires no external hardware

Table of Contents

1. Theory of Operation - The Dallas 1-Wire Protocol

A 1-Wire bus uses only one wire for signaling and power. Communication is asynchronous and half-duplex, and it follows a strict master/slave scheme. One or several slave devices can be connected to the bus at the same time. Only one master should be connected to the bus.

The bus is idle high, so there must be a pull-up resistor present. To determine the value of the pull-up resistor, see the datasheet of the slave device(s). All devices connected to the bus must be able to drive the bus low. A open-collector or open-drain buffer is required if a device is connected through a pin that can not be put in a tri-state mode.

Signaling on the 1-Wire bus is divided into time slots of 60µs. One data bit is transmitted on the bus per time slot. Slave devices are allowed to have a time base that differs significantly from the nominal time base. This however, requires the timing of the master to be very precise, to ensure correct communication with slaves with different time bases. It is therefore very important to obey the time limits described in the following sections.

1.1. Basic Bus Signals

Enter a short description of your topic here (optional).

The master initiates every communication on the bus down to the bit-level. This means that for every bit that is to be transmitted, regardless of direction, the master has to initiate the bit transmission. This is always done by pulling the bus low, which will synchronize the timing logic of all units. There are five basic commands for communication on the 1-Wire bus: “Write 1”, “Write 0”, “Read”, “Reset”, and “Presence”.

“Write 1” signal

A “Write 1” signal is shown in the figure below. The master pulls the bus low for 1 to 15µs. It then releases the bus for the rest of the time slot.

Figure 1-1. "Write 1" Signal

“Write 0” signal

A “Write 0” signal is shown in the figure below. The master pulls the bus low for a period of at least 60µs, with a maximum length of 120µs.

Figure 1-2. "Write 0" Signal

“Read” signal

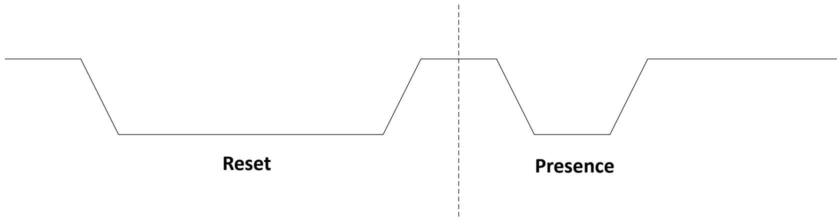
A “Read” signal is shown in the figure below. The master pulls the bus low for 1 to 15µs. The slave then holds the bus low if it wants to send a ‘0’. If it wants to send a ‘1’, it simply releases the line. The bus should be sampled 15µs after the bus was pulled low. As seen from the master’s side, the “Read” signal is in essence a “Write 1” signal. It is the internal state of the slave, rather than the signal itself that dictates whether it is a “Write 1” or “Read” signal.

Figure 1-3. "Read" Signal

"Reset/Presence" signal

A "Reset" and "Presence" signal is shown in the figure below. Note that the time scale is different from the first waveforms. The master pulls the bus low for at least eight time slots, or 480µs and then releases it. This long low period is called the "Reset" signal. If there is a slave present, it should then pull the bus low within 60µs after it was released by the master and hold it low for at least 60µs. This response is called a "Presence" signal. If no presence signal is issued on the bus, the master must assume that no device is present on the bus, and further communication is not possible.

Figure 1-4. "Reset" and "Presence" Signal



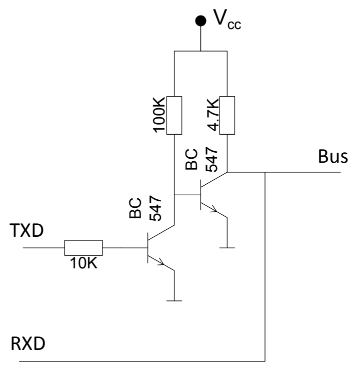
Generating the signals in software

Generating the 1-Wire signals on an AVR in software only is straightforward. Simply changing the direction and value of a general purpose I/O pin and generating the required delay is sufficient. A detailed description is given in the Implementation section.

Generating the signals with a UART

The basic 1-Wire signals can also be generated by a UART. This requires both the TXD and RXD pins to be connected to the bus. An external open-collector or open-drain buffer is required to allow slave devices to pull the bus low when the UART output is high. The figure below shows the connection using NPN-transistors. The resistor values are suggested values only. See the datasheet of the slave device for more information on the recommended pull-up resistance.

Figure 1-5. Open Collector Buffer



The UART data format used when generating 1-Wire signals is eight data bits, no parity, and one stop byte. One UART data frame is used to generate the waveform for one bit or one RESET/PRESENCE sequence. The table below shows how to set up the UART module to generate the waveforms and how to interpret the received data. The corresponding UART bit patterns are shown in Figure 1-6, Figure 1-7, Figure 1-8, Figure 1-9, and Figure 1-10.

Table 1-1. UART Signaling

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.

Figure 1-6. "Write 1" Signal and UART Bit Pattern

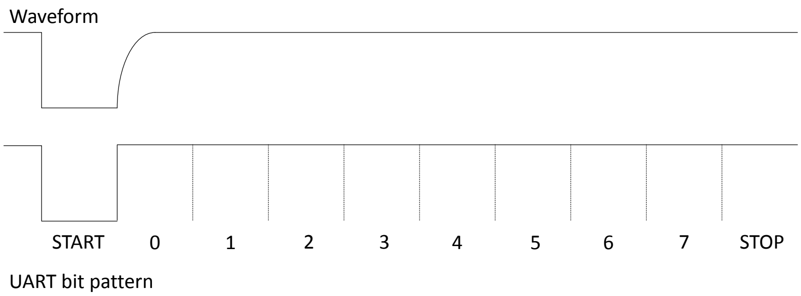


Figure 1-7. "Write 0" Signal and UART Bit Pattern

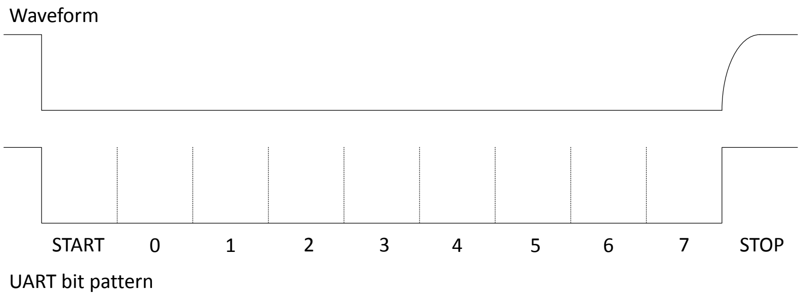


Figure 1-8. "Read 0" Signal and UART Bit Pattern

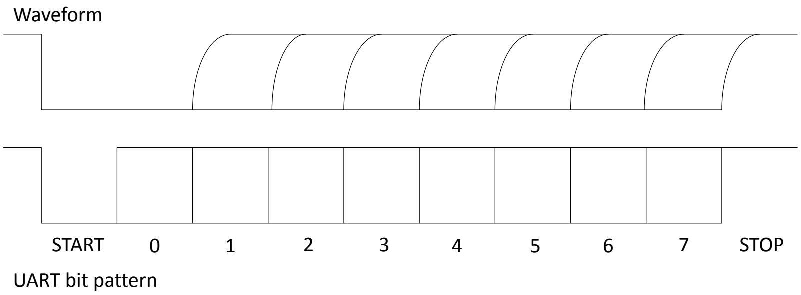


Figure 1-9. "Read 1" Signal and UART Bit Pattern

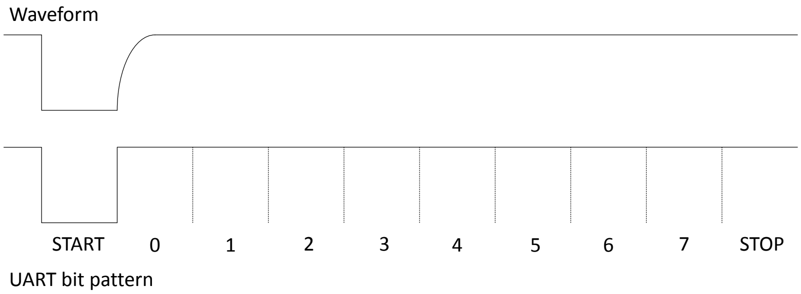
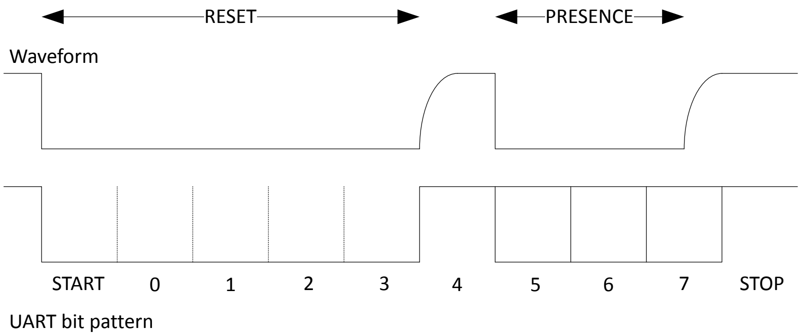


Figure 1-10. Reset/Presence Signal with the UART



1.2. ROM Function Commands

Every 1-Wire device contains a globally unique 64-bit identifier number stored in ROM. This number can be used to facilitate addressing or identification of individual devices on the bus. The identifier consists of three parts; an 8-bit family code, a 48-bit serial number, and an 8-bit CRC computed from the first 56 bits. A small set of commands that operate on the 64 bit identifier are defined. These are called ROM function commands. The table below lists the six defined ROM commands.

Table 1-2. ROM Commands

Command	Code	Usage
READ ROM	33H	Identification
SKIP ROM	CCH	Skip addressing
MATCH ROM	55H	Address specific device
SEARCH ROM	F0H	Obtain IDs of all devices on the bus
OVERDRIVE SKIP ROM	3CH	Overdrive version of SKIP ROM
OVERDRIVE MATCH ROM	69H	Overdrive version of MATCH ROM

READ ROM command

The “READ ROM” command can be used on a bus with a single slave to read the 64-bit unique identifier. If there are several slave devices connected to the bus, the result of this command will be the AND result of all slave device identifiers. Assumed that communication is flawless, the presence of several slaves is indicated by a failed CRC.

SKIP ROM command

The “SKIP ROM” command can be used when no specific slave is targeted. On a one-slave bus, the “SKIP ROM” command is sufficient for addressing. On a multiple-slave bus, the “SKIP ROM” command can be used to address all devices at once. This is only useful when sending commands to slave devices, e.g. to start temperature conversions on several temperature sensors at once. It is not possible to use the “SKIP ROM” command when reading from slave devices on a multiple-slave bus.

MATCH ROM command

The “MATCH ROM” command is used to address individual slave devices on the bus. After the “MATCH ROM” command, the complete 64-bit identifier is transmitted on the bus. When this is done, only the device with exactly this identifier is allowed to answer until the next reset pulse is received.

SEARCH ROM command

The "SEARCH ROM" command can be used when the identifiers of all slave devices are not known in advance. It makes it possible to discover the identifiers of all slaves present on the bus. First the "SEARCH ROM" command is transmitted on the bus. The master then reads one bit from the bus. Each slave places the first bit of its identifier on the bus. The master will read this as the logical AND result of the first bit of all slave identifiers. The master then reads one more bit from the bus. Each slave then places the complement of the first bit of its identifier on the bus. The master will read this as the logical AND of the complement of the first bit of the identifier of all slaves. If all devices have 1 as the first bit, the master will have read 10b. Similarly, if all devices have 0 as the first bit, the master will have read 01b. In these cases, the bit can be stored as the first bit of all addresses. The master will then write back this bit, which in effect will tell all slaves to keep sending identifier bits. If there are devices with both 0 and 1 as the first bit in the identifier on the bus, the master will have read 00. In this case the master must make a choice, whether to continue with the addresses that have 0 in this position or 1. The choice is transmitted on the bus, in effect making all slaves that do not have this bit in this position of the identifier enter an idle state.

The master then goes on to read the next bit, and the process is repeated until all 64 bits are read. The master should then have discovered one complete 64-bit identifier. To discover more identifiers the "SEARCH ROM" command should be run again, but this time a different choice for the bit value should be made the first time there is a discrepancy. Repeating this once for each slave device should discover all slaves. Note that when one search has been performed, all slaves except of one should have entered an idle state. It is now possible to communicate with the active slave without specifically addressing it with the MATCH ROM command.

Overdrive ROM commands

The overdrive ROM commands are not covered here, since overdrive mode is outside the scope of this document, only covering standard speed.

1.3. Memory/Function Commands

Memory/function commands are commands that are specific to one device, or a class of devices. These commands typically deal with reading and writing of internal memory and registers in slave devices. A number of memory/function commands are defined, but all commands are not used by all devices. The order of writes and reads is specific to each device, not part of the general specification. Memory commands will therefore not be covered in detail here.

1.4. Putting it All Together

All 1-Wire devices follow a basic communication sequence:

1. The master sends the "Reset" pulse.
2. The slave(s) respond with a "Presence" pulse.
3. The master sends a ROM command. This effectively addresses one or several slave devices.
4. The master sends a Memory command.

Note: To reach each step, the last step has to be completed. It is, however, not necessary to complete the whole sequence. E.g. it is possible to send a new "Reset" after finishing a ROM command to start a new communication.

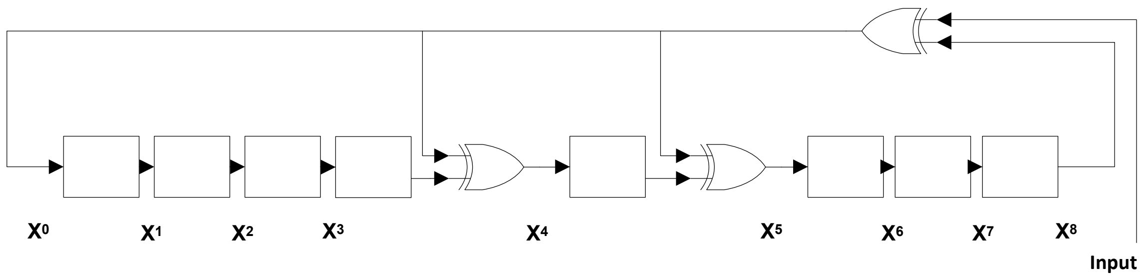
1.5. Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is used by 1-Wire devices to ensure data integrity. The theory behind CRC is outside the scope of this document and will not be further discussed. See "Reference , 2" for more information on CRC.

Two different CRC's are commonly found in 1-Wire devices. One 8-bit CRC (Dallas One Wire CRC, DOW-CRC, or simply CRC8) and one 16-bit CRC (CRC16). CRC8 is used in the ROM section of all devices. CRC8 is also in some devices used to verify other data, like commands issued on the bus. CRC16 is used by some devices to check for errors on larger data sets.

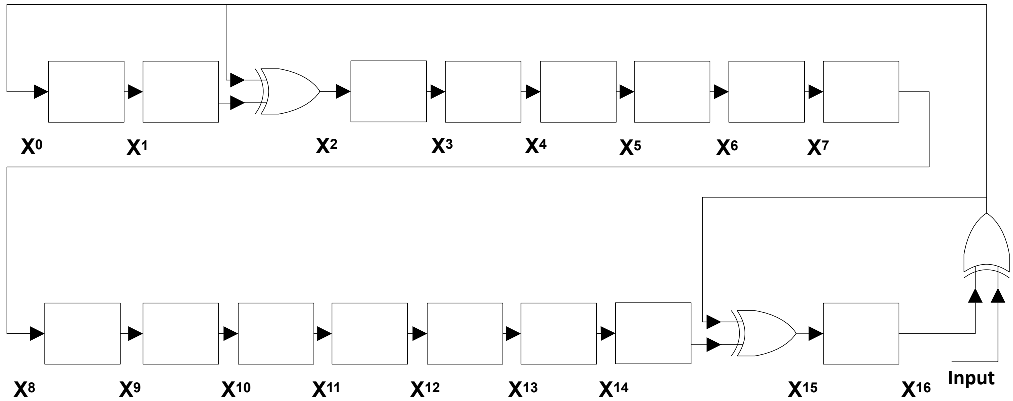
The hardware equivalent of the 8-bit CRC used on the 64-bit identifier is shown in the first figure below. The blocks represent the individual bits in a 8-bit shift register. The equivalent CRC polynomial is $X^8+X^5+X^4+1$

Figure 1-11. Hardware Equivalent of an 8-bit CRC used in 1-Wire Devices



The hardware equivalent of the 16-bit CRC used in some 1-Wire devices is shown in the figure below. The blocks represent the individual bits in a 16-bit shift register. The equivalent polynomial is $X^{16}+X^{15}+X^2+1$.

Figure 1-12. Hardware Equivalent of a 16-bit CRC used in 1-Wire Devices



2. Implementation

Three different 1-Wire implementations are discussed here; software only (polled), polled UART, and interrupt-driven UART. A short description of each is given below. Detailed information about the usage of the drivers is not included in this document. See the documentation included with the source code for this application note for details on how to use the different drivers.

It is possible to implement the 1-Wire protocol in software only, without using any special hardware. This solution has the advantage that the only hardware it occupies is one general purpose I/O pin (GPIO). Since all GPIO pins on the AVR are bi-directional, and have selectable internal pull-up resistors, the AVR can control a 1-Wire bus with no external support-circuitry. In case the internal pull-up resistor is not suitable with the current configuration of slave devices, only one external resistor is needed. On the downside this implementation relies on busy waiting during “Reset/Presence” and bit signaling. To ensure correct timing on the 1-Wire bus, interrupts must be disabled during transmission of bits. The allowed delay between transmission of two bits (recovery time) has no upper limit, however, so it is safe to handle interrupts after every bit transmission. This makes the worst-case interrupt latency due to 1-Wire bus activity equal to execution time of the “Reset/Presence” signal, less than 1ms.

The polled UART driver uses the UART module found on many AVRs to generate the necessary waveforms at the bit-level. The rest of the driver is equal to the software only driver. The main advantage with this driver compared to the software only driver is code size and the fact that interrupts do not need to be turned off during bit signaling since the UART module handles the timing details independently. On the downside it requires two GPIO pins and some external support circuitry.

The interrupt-driven UART driver uses the UART to generate the waveforms in the same way as the polled UART driver. In addition it takes advantage of the interrupt capabilities found in the UART module to automate sending or receiving of up to 255 bits of data.

2.1. Polled Drivers

The polled drivers are divided into two parts. The bit-level waveform generation, and the higher level commands like transmission of bytes and implementation of ROM commands. Only the bit-level procedures are different between the two versions, but they are implemented with a common interface, allowing the higher level commands to be used with either driver.

2.1.1. Software Only Implementation

With the software only implementation provided with this application note, it is possible to have several 1-Wire buses connected to one AVR. All buses must, however, be connected to the same I/O port, but which port is optional at compile-time. This limits the number of buses to eight, but placement of buses within the port is fully configurable. All pins not used for 1-Wire buses are unaffected. Since all 1-Wire buses are connected to the same port, several operations can be performed on one or more buses at the same time. This is accomplished through an argument called pin or pins, that is passed to every function. This argument should contain a bit-mask of the pins that should be used for this operation. It is for instance possible to send the Reset signal to eight buses at the same time by passing 0xff as the pins argument. The value returned from the same function will be a bit-mask of all buses where one or more slave devices answered with a presence signal. This bit mask can then be passed as the pins argument to a function issuing the SKIP ROM command, and so on. All functions in this implementation supports pin selection. As a general rule, all commands that write to the bus can address several buses at the same time. Commands that read more than one bit from the bus in some way can only address one bus.

Initialization

The initialization procedure for the software only 1-Wire interface is really simple. It consists only of setting the 1-Wire pins in input mode, and enable the internal pull-up resistor, if required, to put the bus in idle mode. Some devices will react to this rising edge on the bus as the end of a Reset signal and reply with a Presence signal. To ensure that this signal does not interfere with any communication, a delay equally long to the reset recovery time is inserted.

Bit-level functions

The bit-level functions are implemented according to application note AN126 from Dallas Semiconductors. All timing parameters comply with the recommended values in this application note. Ten different delays are needed. These are listed in the table below.

Table 2-1. Bit Transfer Layer Delays

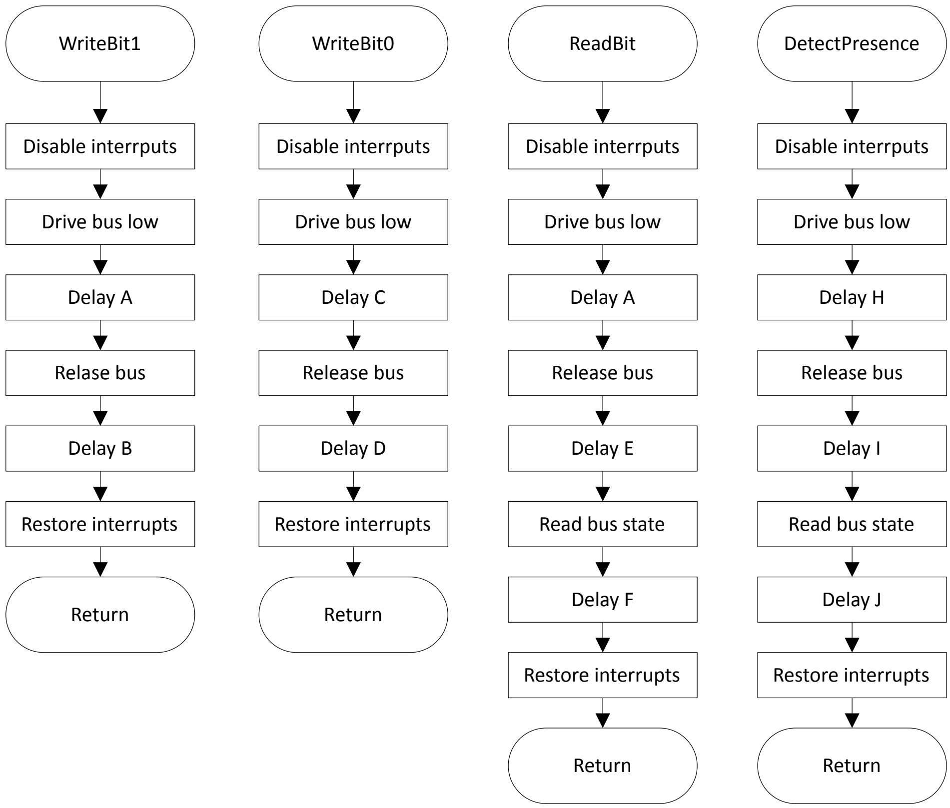
Parameter	Recommended delay [µs]
A	6
B	64
C	60
D	10
E	9
F	55
G	0
H	480
I	70
J	410

Note: G delay is zero in standard mode.

Since the I/O operations are implemented in C and not assembly language, compiler optimizations and other factors could affect timing. It is recommended to observe the waveforms generated by each bit-level function with an oscilloscope, and adjust delays if needed.

The bit transfer layer functions are implemented as shown in the figure below. Note that the function “DetectPresence” both sends the “Reset” signal, and listens for the “Presence signal”. Note that all bit transfer layer functions can address several buses at the same time.

Figure 2-1. Bit Transfer Layer Functions



Two macros are included to drive the bus low and to release the bus. These are implemented as macros because they occur frequently, and the overhead caused by function calls is unwanted because of the strict timing requirements.

2.1.2. Polled UART Implementation

In this implementation, all the timing details are taken care of by the UART module. To send a bit, the UART baud rate is set to the appropriate value, and the UART data register is loaded with a value that will generate the desired waveform as described in the “Generating the signals with a UART” section.

Initialization

To initialize the 1-Wire interface for the polled UART driver, the UART module has to be initialized with the right parameters. Enable transmission and reception, set data format to eight bits, no parity, one stop bit and set baud rate to 115.2kBaud.

This will cause the TXD pin to enter a UART idle state, which is a logic high. Slave devices will interpret this rising edge as the end of a RESET signal, and answer with a presence signal.

Bit-level functions

All bit-level functions in the Polled UART driver are implemented through one common function called OWI_TouchBit. This function outputs the first input argument to the UART module, waits until UART reception is complete, and then returns the AVR318112579A-AVR-09/04 received value. Each of the bit-level functions calls OWI_TouchBit with the value that will generate the correct waveform on the bus.

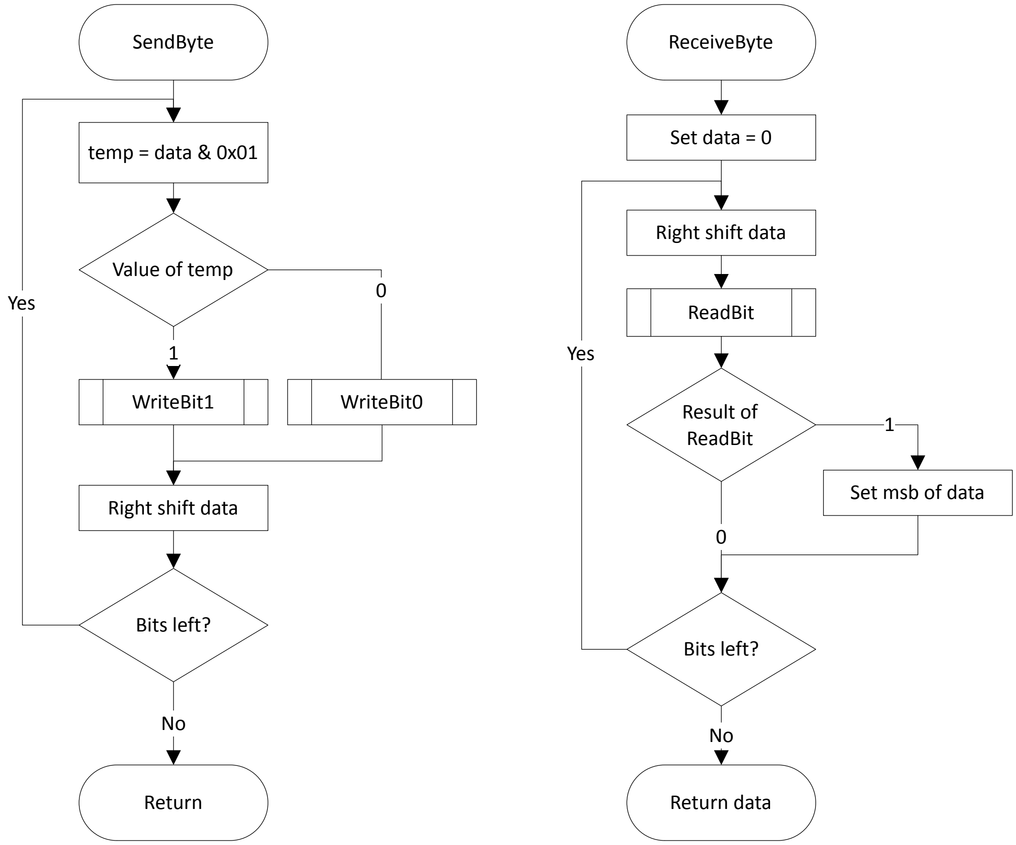
The interface to these functions is the same as for the software only implementation. The ‘pins’ argument is, however, not necessary in the polled UART driver. A set of macros makes it possible to call these functions with or without the pins argument. If the pins argument is included, it will be removed by the macros.

2.1.3. Higher Level Functions

Note that many functions in this layer accept an argument of type unsigned char pointer. This pointer should point to an array of eight bytes of memory that can be used by the function. Allocation, and sometimes initialization, of these arrays must be done by the caller. This document clearly states when the memory has to be initialized in a special way before calling a function.

2.1.3.1. Byte Transmission Functions

Figure 2-2. Byte Transmission Functions



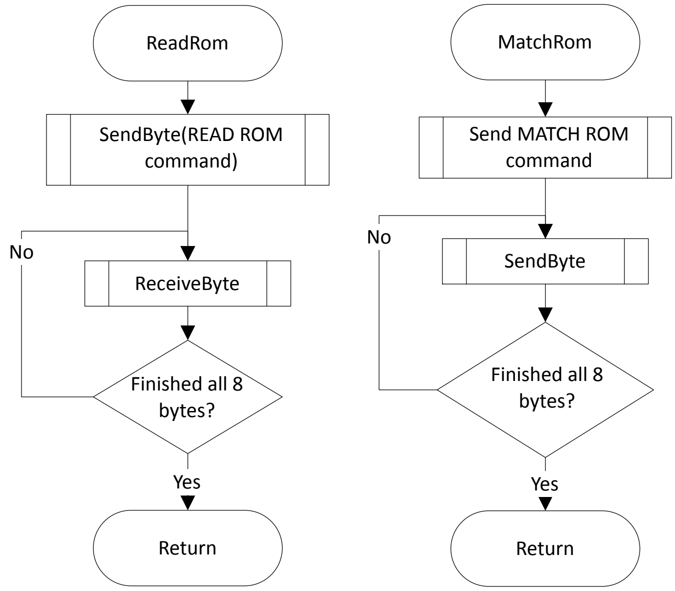
2.1.3.2. ROM Commands

All general ROM commands for standard speed communication are implemented.

The simplest ROM command is the SKIP ROM command. It simply calls the SendByte function with the SKIP ROM command byte as argument.

Flowcharts for the READ ROM and MATCH ROM commands are shown in the figure below.

Figure 2-3. Read ROM Flowchart and Match ROM Flowchart



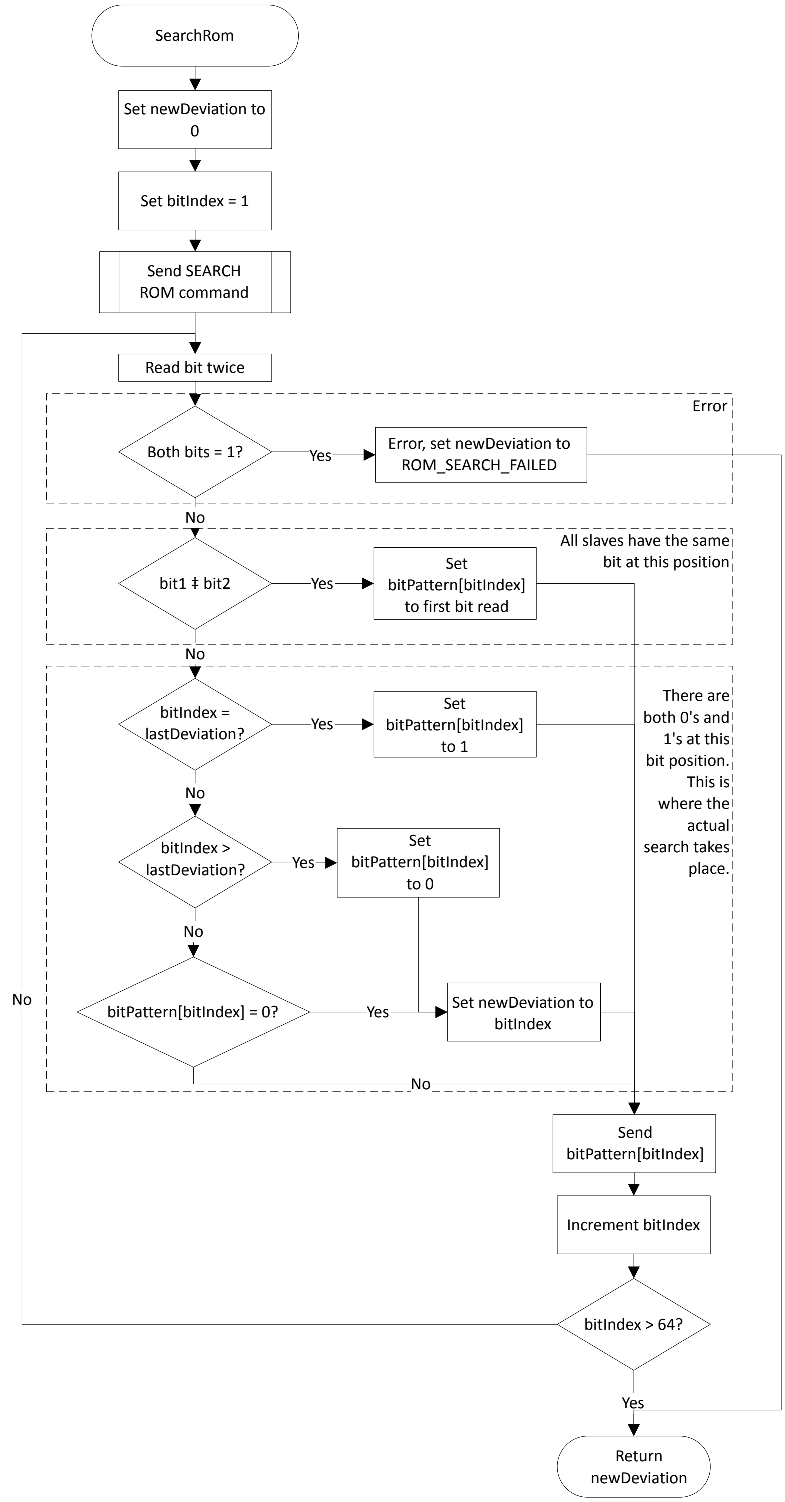
The flowchart for the SEARCH ROM command is shown in the figure below. This function will find one slave device for each time it is run, until there are no undiscovered slave devices on the bus. The last time it is run, it will return OWI_ROM_SEARCH_FINISHED. In addition to the 'pin' parameter, used to select which bus to perform the search on, two parameters must be passed to this function: 'lastDeviation' and 'bitPattern'. These parameters control the slave device search. Refer to the table below to understand how to use these parameters to complete a full search for all slave devices.

Table 2-2. bitPattern and lastDeviation Usage

	BitPattern	lastDeviation
First time	Zero filled eight byte array	0
Consecutive runs	A copy of the eight byte array returned through bitPattern pointer last run	Value returned from SearchRom last run

The function is implemented in this way to give the caller maximum flexibility. The example software for the polled driver shows how it can be used to implement the full search.

Figure 2-4. Search ROM Command



2.1.4. Timing Considerations

It is important to be able to generate the waveforms as precisely as possible. To do this, precise delays are needed. The number of clock cycles needed to delay for a certain number of microseconds is computed at compile time. When generating waveforms, some clock cycles are lost when pulling the bus low and when releasing the bus. These clock cycles are subtracted from the number of clock cycles needed to generate the delay. If the clock frequency is too low, this could generate a negative delay. A clock frequency higher than 2.17MHz is needed to be able to generate the shortest delays.

2.1.5. Interrupt-driven UART Implementation

The interrupt-driven UART driver has the same hardware requirement as the polled UART driver.

The basic functionality of the interrupt-driven implementation presented in this application note is to automate transmission and reception of larger chunks of data on the bus. This is done in two Interrupt Service Routines (ISRs). A set of helper functions can be called to set up all the necessary parameters, and these ISRs completes the transaction automatically. It is possible to do a Reset/Presence sequence or transfer anywhere between 1 and 255 bits of data in one direction without intervention.

To make the ISRs as simple as possible, they do not differentiate between transmission and reception. The UDRE ISR simply sends one bit from the data buffer every time it is run. The RXC ISR receives the same bit, and puts it back into the data buffer no matter which direction data was sent. During transmission, the data sent will be identical to the data received, and the data buffer remains unchanged. During reception, only ‘1’s should be transmitted, since the ‘write 1’ waveform is the same as the read waveform. The signal is sampled to find the value written by the slave device. This value is then placed in the data buffer.

Three global flags signal the state of the 1-Wire driver; busy, presence, and error. The busy flag is set as long as there is more data to transfer. The presence flag is set if a Presence signal is detected when sending a Reset signal. This flag remains set until a Reset signal on the bus does not return a Presence signal. The error flag is set when the UART receiver detects a frame error. In this situation, a new Reset signal should be transmitted on the bus. This will reset all slaves on the bus, as well as the internal state of UDRE and RXC ISRs.

As ISRs should be executed as quickly as possible, more complex functions like ROM commands are not implemented in the ISRs. The included example code shows how such behavior could be implemented in a Finite State Machine (FSM).

2.1.5.1. The Interrupt Service Routines

Flowcharts for the ISRs are shown in the two figures below. The UART Data Register Empty (UDRE) ISR is run every time there is room for data in the UART transmission buffer. The UART Receive Complete (RXC) ISR is run every time data has been received and is ready in the UART reception buffer. Flowcharts for the helper functions are shown in figure Helper Functions.

Figure 2-5. UDRE Interrupt Service Routine

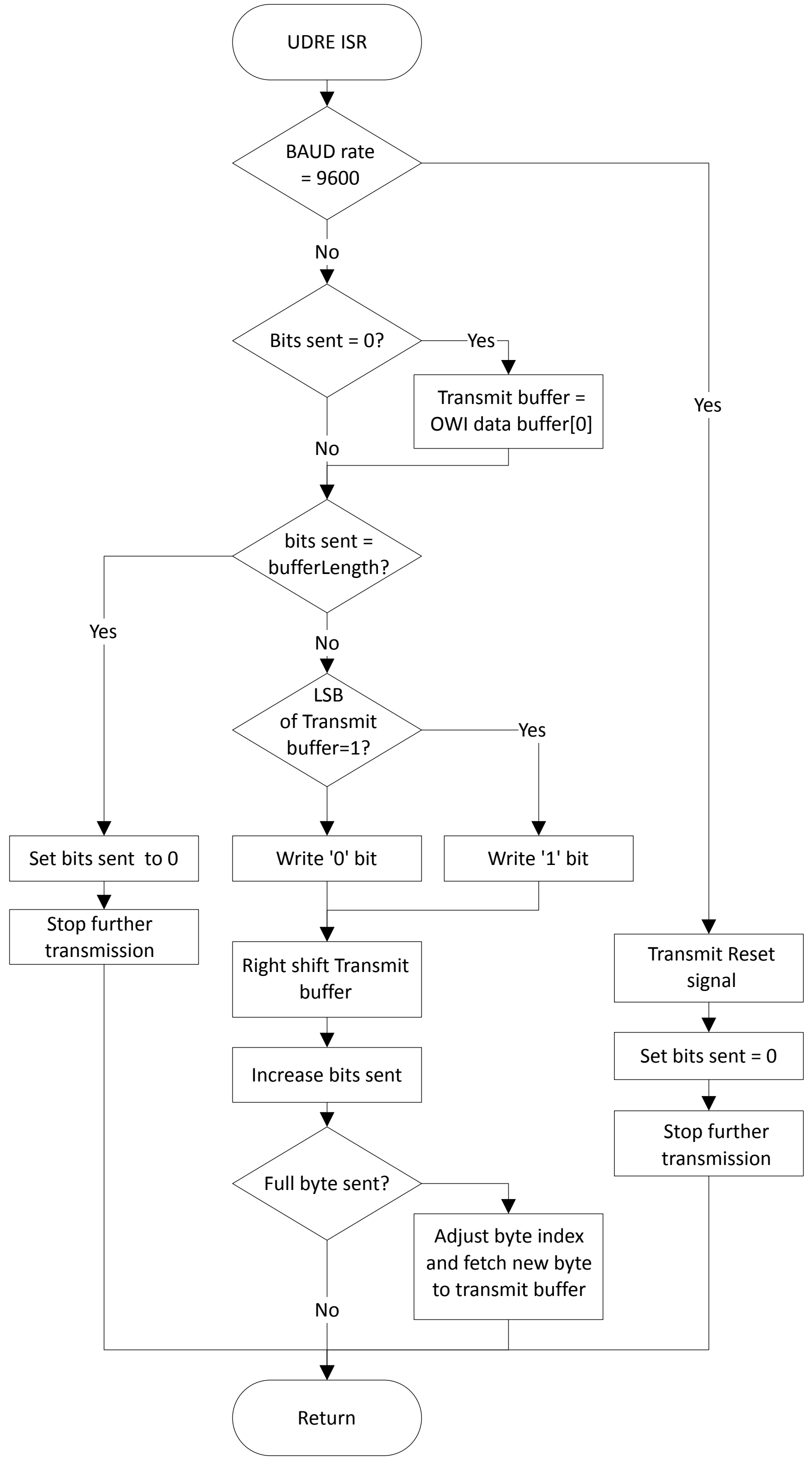
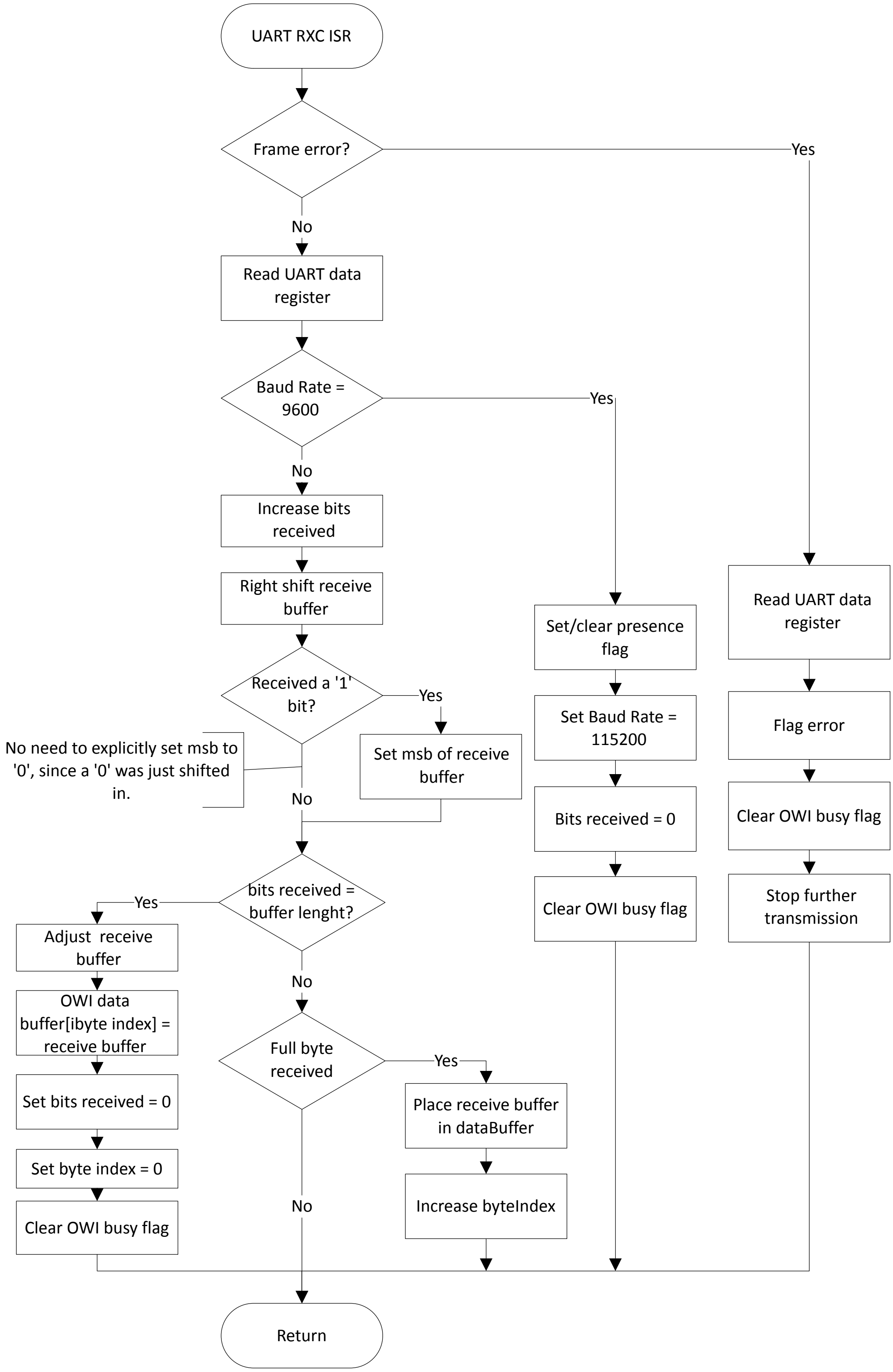


Figure 2-6. RXC Interrupt Service Routine



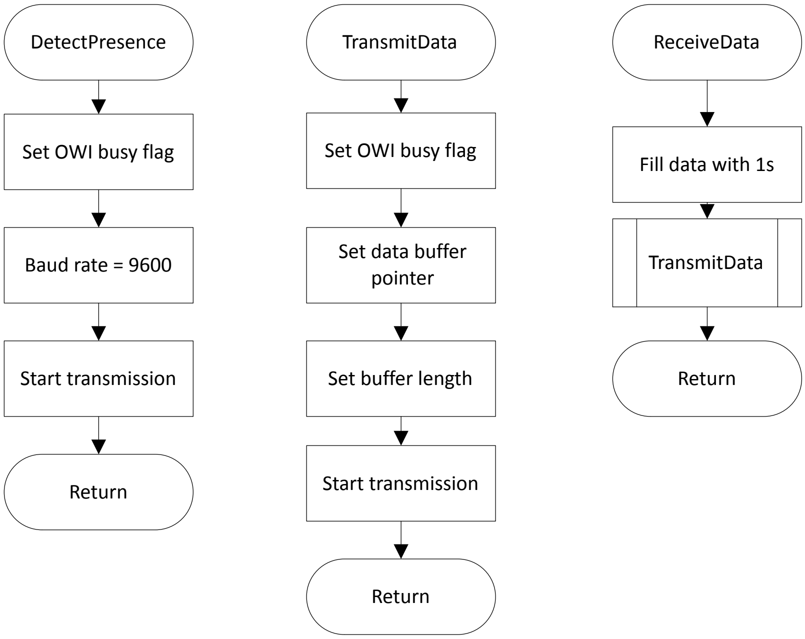
2.1.5.2. Helper Functions

The helper functions set up some parameters that are necessary for the automated interrupt-driven transmission to succeed. After setting up all the necessary parameters, transmission is initiated by enabling the UDRE interrupt.

Flowcharts for the helper functions are shown in the figure below.

Note that the ReceiveData function actually fills the data buffer with ‘1’s and calls the TransmitData function. The RXC ISR will sample the signal and place the value read from the slave device into the data buffer.

Figure 2-7. Helper Functions



2.2. CRC Computation

The algorithm used to compute the two different CRC’s are described below.

The CRC is either set to 0, or to a CRC “seed”. This is explained below.

1. Find the "logical exclusive or" (XOR) between the lsb of the CRC and the lsb of the data.
2. If this value is 0-Right shift CRC.
3. If the value was 1:
   - 3.1. Find the new CRC value by taking the "logical exclusive or" (XOR) of the CRC and the CRC polynomial.
   - 3.2. Right shift CRC.
   - 3.3. Set the msb of the CRC to 1.
4. Right shift the data.
5. Repeat the complete sequence eight times.

This algorithm can be used to compute both CRC8 and CRC16. The only difference is the width of the CRC shift register (8 bits for CRC8, 16 bits for CRC16) and the value of the polynomial. This number will simulate the connection of the XOR gates in hardware. The value of the polynomial is 18h for CRC8 and 4002h for CRC16.

The algorithms are implemented to find the CRC value of one byte at a time, but a CRC “seed” can be passed as an argument to the CRC routines. In this way the result of one CRC operation can be passed to the next one along with the next byte, in effect computing the CRC of an arbitrary number of bytes.

CRC checking of 64-bit identifiers are implemented in OWI_CheckRomCRC. It simply computes the CRC8 value of the first 56 bits, and compares it to the last 8 bits of the identifier.

2.3. Code Examples

The two code examples included shows how to use the different implementations of the 1-Wire driver.

2.3.1. Polled Example

The code example for the polled drivers will search the buses defined by “BUSES” for devices. The devices are stored in an array of type OWI_device. OWI_device is a struct containing information about what bus a device is connected to and its 64-bit identifier. The driver then searches through the available slave devices for a DS1820 temperature sensor on PORTD0. If the device is found on the bus, it will constantly be negotiated in an eternal loop. In each iteration, the temperature of the DS1820 is polled and the temperature is output to PORTB, so it can be observed for instance on the LED’s of an STK®600 development board.

This code example is intended to show how the different parts of the driver can be used. The code is very general, and not optimized for the objective. Note that because of this, the code example will not fit on a device with less than 4kB of program memory. The driver is, however, fully compatible with all AVR microcontrollers, including 1KB devices.

2.3.2. Interrupt-driven Example

In the interrupt-driven example, a finite state machine (FSM) is implemented. If the driver is not busy transmitting data on the bus, this FSM is called from an eternal loop. When the driver is busy, the FSM will be skipped to allow any other code to be run. The FSM itself assumes that there is a sole DS1820 temperature sensor available on the bus. It will read the current temperature, and compute the CRC to make sure that it was read correctly. The temperature is then put in a global variable. Whenever the driver is busy, the eternal loop outputs the temperature to PORTB, so it can be observed for instance on the LEDs of an STK600 development board.

3. Getting Started

This chapter outlines how to get started with the example code included with this application note.

3.1. The Source Code

The example code is written for Atmel START. It can be downloaded from "BROWSE EXAMPLES" entry of Atmel START for both Atmel Studio 7 and IAR IDE.

Double click the downloaded .atzip file and project will be imported to Atmel Studio 7. To import the project in IAR™, refer "Atmel START in IAR" , select Atmel Start Output in External Tools → IAR.

3.1.1. Polled Driver

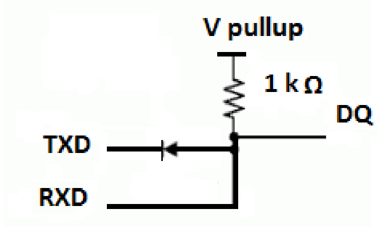
A short description of each file in the polled driver is shown in the table below.

Table 3-1. Polled Driver Files

File	Contains
main.c	Code example for the polled driver
OWISWBitFunctions.c	Implementation of the software only bit-level functions
OWIUARTBitFunctions.c	Implementation of the UART bit-level functions
OWIBitFunctions.h	Common header file for OWISWBitFunctions.c and OWIUARTBitfunctions.c
OWIHighLevelFunctions.c	High level functions
OWIHighLevelFunctions.h	Header file for OWIHighLevelFunctions.c
OWIPolled.h	Configuration header file for the polled drivers
Documentation of the source code in this folder	source.doc

- Open the Atmel Studio 7 project or IAR project. (After downloading .atzip from Atmel START and importing in Atmel Studio 7 or IAR.)
- Open the file “OWIPolled.h” for editing and locate the section named “User defines”
- Choose between 'software only' or 'UART driver' by uncommenting one of the lines as described in the file
- Move down to the section corresponding to the selected driver
- Adjust the defines in the section according to the hardware setup as described in the file
- The project is now ready to be compiled
- Driver mode can be selected as OWI_SOFTWARE_DRIVER or OWI_UART_DRIVER from OWIPolled.h file. For OWI_UART_DRIVER mode , Open drain circuit needs to be connected at TXD and RXD pins as shown in the figure below. DQ is 1 wire interface from 1-Wire device.

Figure 3-1. Open Drain Circuit



3.1.2. Interrupt-driven Driver

A short description of each file in the interrupt-driven driver is shown in the table below.

Table 3-2. Interrupt Driver Files

File	Contains
main.c	Code example for the interrupt-driven driver
OWIInterruptDriven.h	Configuration header file for the interrupt-driven driver
OWIIntFunctions.c	Implementation of the interrupt-handlers and helper functions
OWIIntFunctions.h	Header file for OWIIntFunctions.c
source.doc	Documentation of the source code in this folder

To get started with the interrupt-driven driver, follow the steps below:

- Open the Atmel Studio 7 project or IAR project. (After downloading .atzip from Atmel START and importing in Atmel Studio 7 or IAR.)
- Open the file "OWIInterruptDriven.h" for editing and locate the section named "User defines"
- Change the defines in the "User defines" section to reflect the hardware setup
- The project is now ready to be compiled
- Open drain circuit needs to be connected at TXD and RXD pins as shown in the figure Open Drain Circuit

4. References

1. Application note 126, 1-Wire communication through software, Dallas Semiconductors, 2004.
2. Book of iButton standards, Dallas Semiconductors, 1997.
3. Application note 214, Using a UART to implement a 1-wire bus master, Dallas Semiconductors, 2002.

5. Revision History

Doc. Rev.	Date	Comments
2579B	10/2016	Atmel START code release
2579A	09/2004	Initial document release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**