AVR 300: Software I²C™ Master Interface

Features

- **Uses No Interrupts**
- **Supports Normal And Fast Mode**
- **Supports Both 7-Bit and 10-Bit Addressing**
- **Supports the Entire AVR Microcontroller Family**

Introduction

The need for a simple and cost effective inter-IC bus for use in consumer, telecommunications and industrial electronics, led to the developing of the I²C bus. Today the I²C bus is implemented in a large number of peripheral and microcontrollers, making it a good choice in low speed applications.

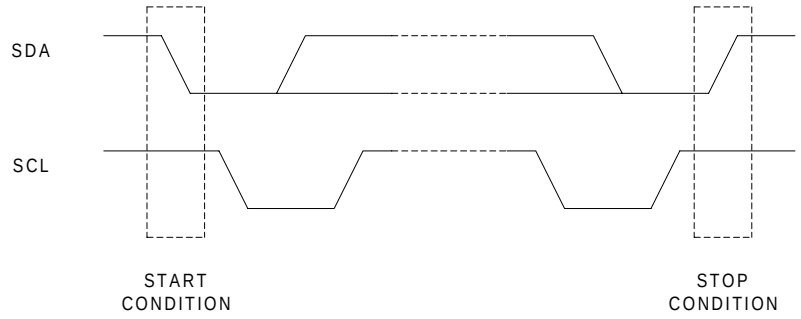
The AVR microcontroller family does not have dedicated hardware for I²C operation, but because of the flexible I/O and high processing speed, an efficient software I²C single master interface, can easily be implemented.

Theory of Operation

The I²C bus is a two-wire synchronous serial interface consisting of one data (SDA) and one clock (SCL) line. By using open drain/collector outputs, the I²C bus supports any fabrication process (CMOS, bipolar and more).

The I²C bus is a multi-master bus where one or more devices, capable of taking control of the bus, can be connected. When there is only one master connected to the bus, this does not need to support handling of bus contentions and inter master access (a master accessing another master). Only master devices can drive both the SCL and SDA lines while a slave device is only allowed to issue data on the SDA line.

Figure 1. START and STOP conditions



Data transfer is always initiated by a bus master device. A **high** to **low** transition on the SDA line while SCL is high is defined to be a START condition (or a repeated start condition). A START condition is always followed by the (unique) 7-bit slave address and then by a data direction bit. The slave device addressed now acknowledges to the master by holding SDA low for one clock cycle. If the master does not receives any acknowledge, the transfer is terminated. Depending of the data direction bit, the master or slave now transmits 8-bit of data on the SDA line. The receiving device then acknowledges the data. Mul-

8-Bit Microcontroller

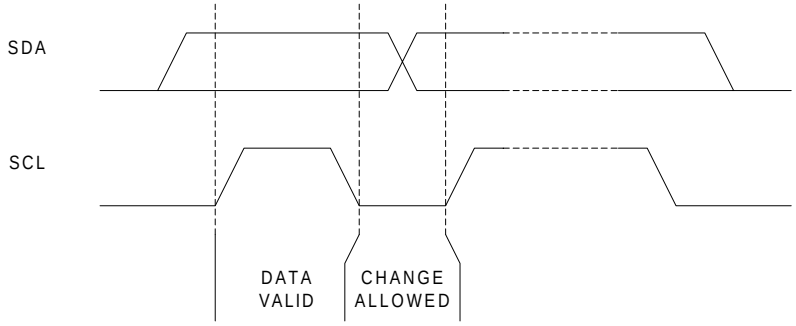
Application Note

0954A-A–9/97

tiple bytes can be transferred in one direction before a repeated START or a STOP condition is issued by the master. The transfer is terminated when the master issues a STOP condition. A STOP condition is defined by a **low** to **high** transition on the SDA line while the SCL is high.

If a slave device cannot handle incoming data until it has performed some other function, it can hold SCL low to force the master into a wait-state.

Figure 2. Bit transfer on the I^2C bus



Change of data on the SDA line is only allowed during the low period of SCL as shown in Figure 2. This is a direct consequence of the definition of the START and STOP conditions. A more detailed description and timing specifications, can be found in [1].

Transferring Data

All transfer on the bus is byte sized. Each byte is followed by an acknowledge bit set by the receiver. The slave address byte contains both a 7-bit address and a read/write bit.

Figure 3. Byte Transfer Formats

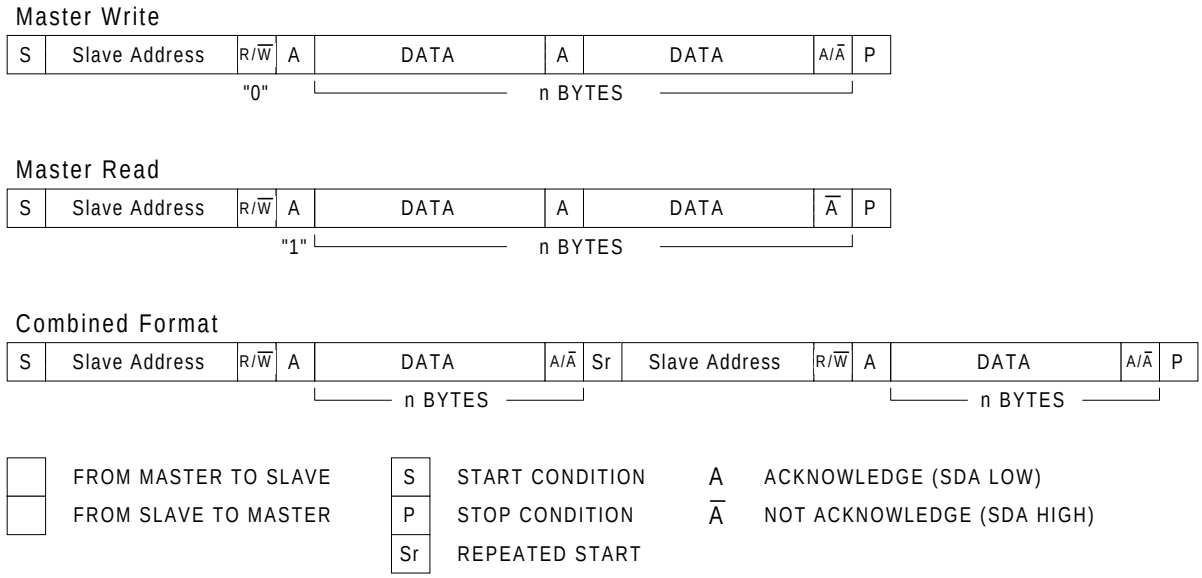


Figure 3 shows the valid data transfer formats. In the combined format, multiple data can be sent in any direction (to the same slave device). A change in data direction is done by using a *repeated* START condition. Note that a master read operation must be terminated by not acknowledging the last byte read.

Connection

Both I^2C lines (SDA and SCL) are bi-directional therefore outputs must be of an open-drain or an open-collector type. Each line must be connected to the supply voltage via a pull-up resistor. A line is then logic high when none of the connected devices drives the line, and logic low if one or more is drives the line low.

Figure 4. Physical connection to the I^2C bus

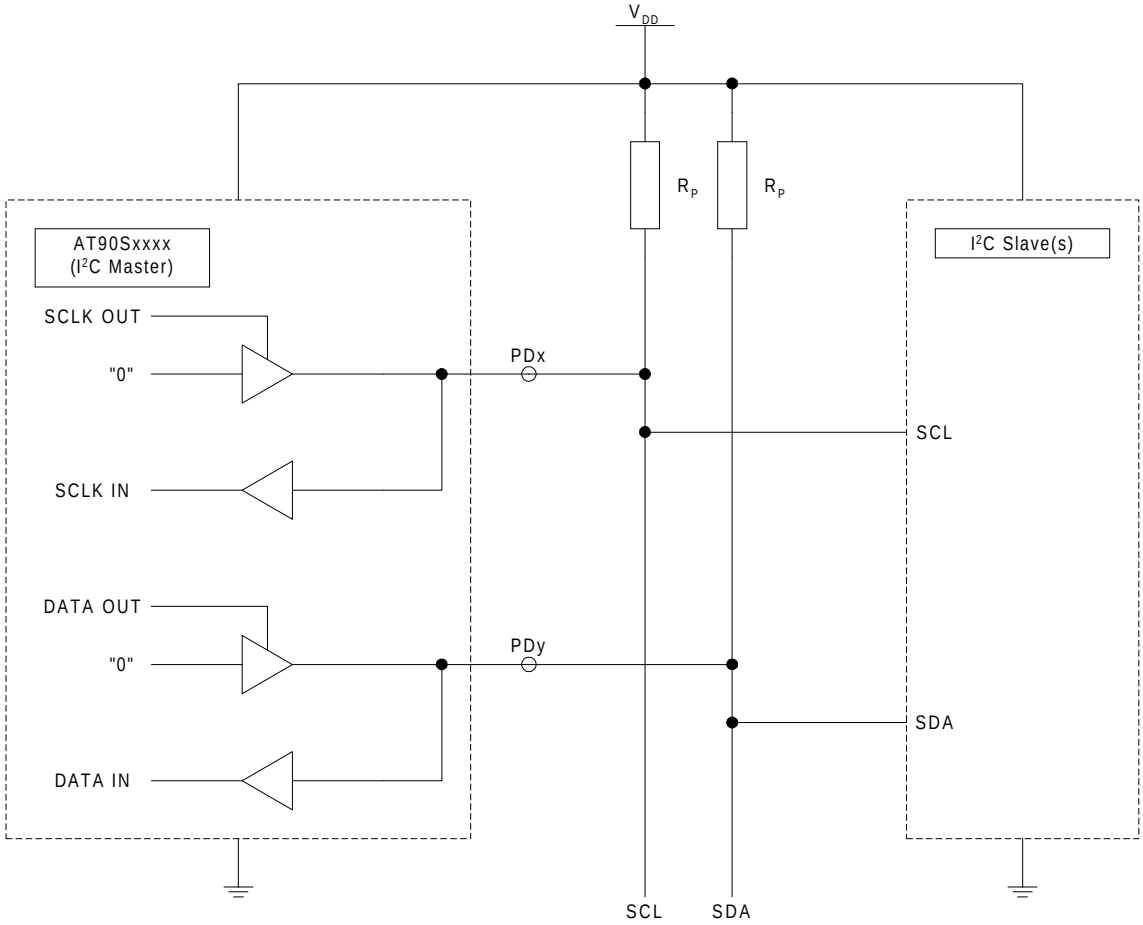


Figure 4 shows how to connect the AVR microcontroller to the I^2C bus. The value of R_P depends on V_{DD} and the bus capacitance (typically 4.7k).

Implementation

The only resources used by the I^2C master routines presented in this application, is the two pins for SCL and SDA on port D. Since the I^2C bus is synchronous, the duty cycle and the period time to the serial clock line (SCL) is not critical. Therefore it is not necessary to 'fine-tune' the routines which would cause to an increase of program size.

There are two types of delays used in this implementation: quarter period and half period delays. For I^2C in normal mode (100 kHz), these delays must be $t_{quarter} > 2.5$ µs and $t_{half} > 5.0$ µs. For I^2C in fast mode (400 kHz) the parameters are $t_{quarter} > 0.6$ µs and $t_{half} > 1.3$ µs.

There is a large number of possible implementations of the delay loop. All the implementations depends on the MCU clock frequency. It is not possible to make a generalized version which is efficient for all clock speeds.

The following steps show how to choose the most efficient implementation of the delays:

1. Calculate the required number of clock cycles both delays:

   $n = t * fosc$; $t = t_{quarter}$ and t_{half}, fosc is MCU clock.

2. Use n to chose one of the following methods for both delays.

n	Delay method
< 1	remove all calls to the delay routine
1 < n < 2	replace all calls to the delay routine with one ‘nop’ instruction
2 < n < 3	replace all calls to the delay routine with an ‘rjmp 1’ instruction
2 < n < 7	the delay routine should consist of one ‘ret’ instruction only

> 7	use the following routine :
	`ldi i2cdelay, 1+ (n-7)/3`
	`loop: dec i2cdelay`
	`brne loop`
	`ret`
	`(this routine is used in the program code!)`

I2C Subroutines

‘i2c_init’

Initializes SCL and SDA lines. The SCLP and SDAP constants located top of the program code, chooses the pin number on port D. It is possible to use any pins on any port by changing the program code if required.

All the port D initialization can be put in this subroutine to reduce code size.

‘i2c_start’

Generate start condition and sends slave address. All data transfer must start with this subroutine. When a transfer is done the ‘i2c_end’ must be called. *When the bus is free (after ‘i2c’_end is called) all registers are free for other usage.*

Parameter	Value
Code Size	3
Execution cycles	N/A
Register Usage	Low registers :None High registers :3 Global registers :1

Register	Input	Internal	Output
r16		‘i2cdelay’ - Delay loop counter	
r17		‘i2cdata’ - Transmit buffer	
r18	‘i2cadr’ - Slave address and transfer direction (global)		

‘i2c_rep_start’
Generate repeated start condition and sends slave address. A repeated START can only be given after a byte has been read or written.

Parameter	Value
Code Size	5
Execution cycles	N/A
Register Usage	Low registers :None High registers :3 Global registers :1

Register	Input	Internal	Output
r16		‘i2cdelay’ - Delay loop counter	
r17		‘i2cdata’ - Transmit buffer	
r18	‘i2cadr’ - Slave address and transfer direction (global)		

‘i2c_write’
Writes data (one byte) to the I^2C bus. This function is also used for sending the address.

Parameter	Value
Code Size	16
Execution cycles	N/A
Register Usage	Low registers :None High registers :2 Global registers :None

Register	Input	Internal	Output
r16		‘i2cdelay’ - Delay loop counter	
r17	‘i2cdata’ - Data to be written		

‘i2c_get_ack’
Get slave acknowledge response. The reason for separate this subroutine from the ‘i2c_write’ routine is to get a more readable program code.

Parameter	Value
Code Size	11
Execution cycles	N/A
Register Usage	Low registers :None High registers :1 Global registers :None

Register	Input	Internal	Output
r16		‘i2cdelay’ - Delay loop counter	

Figure 5. ‘i2c_start’, ‘i2c_rep_start’, ‘i2c_write’ and ‘i2c_get_ack’ Flow Chart

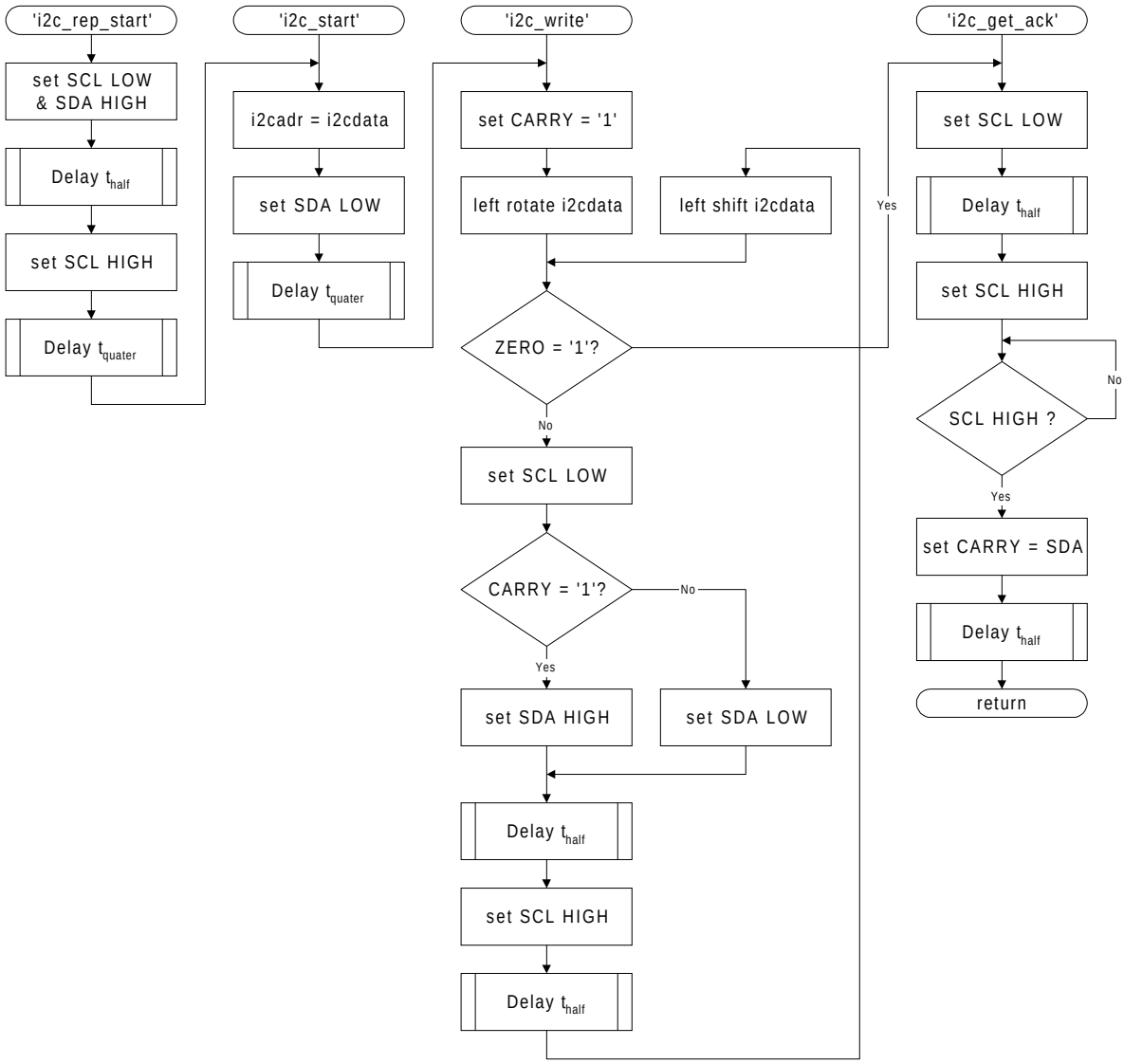


Figure 5 shows the flow chart for ‘i2c_start’, ‘i2c_rep_start’, ‘i2c_write’ and ‘i2c_get_ack’. These subroutines shares program code to reduce size.

‘i2c_read’
Reads data (one byte) from the I^2C bus.

Parameter	Value
Code Size	11
Execution cycles	N/A
Register Usage	Low registers :None High registers :3 Global registers :1

Register	Input	Internal	Output
r16		'i2cdelay' - Delay loop counter	
r17			'i2cdata' - Received data
r19		'i2cstat' - Store acknowledge bit (global)	

'i2c_put_ack'

Put an acknowledge bit depending on carry flag is set or not. Separating this subroutine from the 'i2c_read' routine is convenient for the user if a acknowledge is based on the result of the read operation.

Parameter	Value
Code Size	12
Execution cycles	N/A
Register Usage	Low registers :None High registers :2 Global registers :1

Register	Input	Internal	Output
r16		'i2cdelay' - Delay loop counter	
r19		'i2cstat' - Acknowledge bit (global)	

Figure 6. ‘i2c_read’ and ‘i2c_put_ack’ Flow Chart

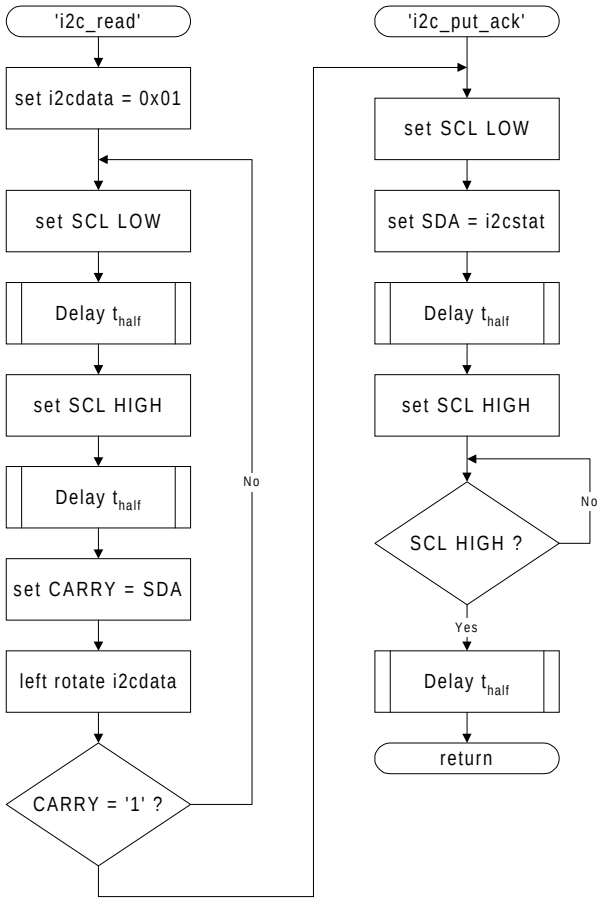


Figure 5 shows the flow chart for ‘i2c_read’ and ‘i2c_put_ack’. These subroutines shares program code to reduce size.

‘i2c_stop’
Generate stop condition. When a transfer is done the ‘i2c_end’ must be called. When the bus is free (after ‘i2c’_end is called) all registers are free for use.

Parameter	Value
Code Size	8
Execution cycles	N/A
Register Usage	Low registers :None High registers :1 Global registers :None

Register	Input	Internal	Output
r16		‘i2cdelay’ - Delay loop counter	

Figure 7. ‘i2c_stop’ Flow Chart

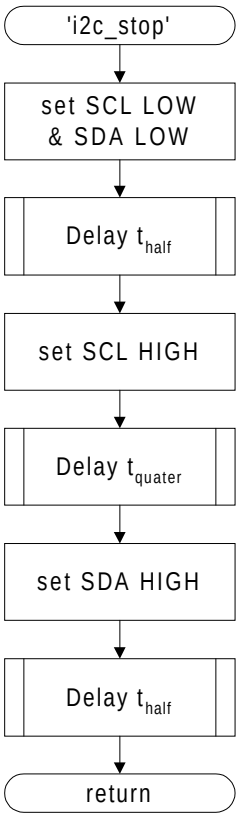


Figure 7 shows the flow chart for ‘i2c_stop’.

‘i2c_do_transfer’
The ‘i2c_do_transfer’ routine is implemented for convenience only. It uses the direction bit from the last address byte send, to decide whether to call the ‘i2c_read’ or the ‘i2c_write’ routine.

Tips and Warnings

The main loop in the program shows an example of reading and writing data to a 256byte SRAM. This is a simple demonstration of how to use the I^2C routines. Typically the reading and writing of SRAM will be implemented as functions calls, but since there is a large variety of slave implementations and ways of accessing them, the making this type of function calls is left for the user.

Warning! Do not change the order of the I^2C routines. Most routines expects to be followed by a another specific I^2C routine to work correctly.

Conclusion

This application note shows how to implement a master I^2C interface on any AVR microcontroller device. This by using a minimum of resources. Since no interrupts are used in the implementation, these are free for other applications. It is also possible to use the I^2C interface inside interrupts.

References

[1] The I^2C-bus and how to use it (including specifications), Philips Semiconductors, April 1995.