AVR302: Software I^{2}C™ Slave Implementation

Software I^{2}C Slave Implementation

Application Note

Features

- **Interrupt Based**
- **Device can be Given Any 7-bit Address (expandable to 10-bit)**
- **Supports Normal and Fast Mode (400 kbps)**
- **Easy Insertion of “Wait States”**
- **Supports Wake-up from Idle Mode**
- **Code Size 160 Words (maximum)**

Introduction

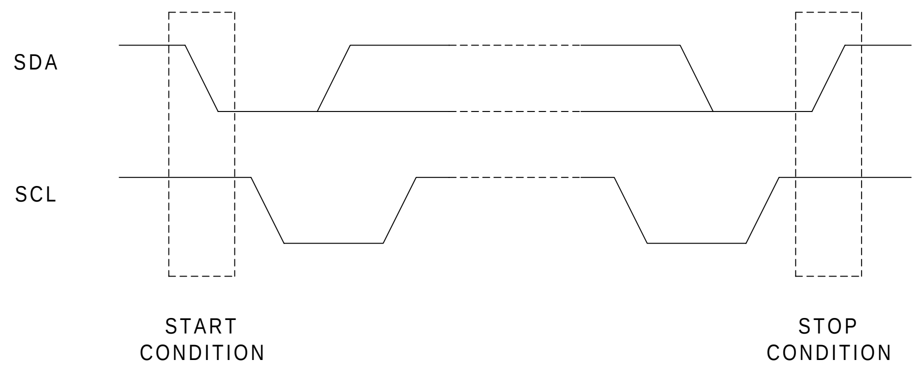
The need for a simple and cost effective inter-IC bus for use in consumer, telecommunications and industrial electronics, led to the developing of the I^{2}C bus. Today the I^{2}C bus is implemented in a large number of peripheral and microcontrollers, making it a good choice in low speed applications. The AT90S1200 does not have dedicated hardware for the I^{2}C, but because of the high processing speed and flexible I/O ports, an effective software I^{2}C slave implementation, can easily be done. The AT90S1200 is the only 8-bit MCU known to date that can perform fast (400 kbps) I^{2}C slave operations in software.

Theory of Operation

The I^{2}C 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 I^{2}C bus supports any fabrication process (CMOS, bipolar and more).

The I^{2}C bus is a multi-master bus where one or more devices, capable of taking control of the bus, can be connected. 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

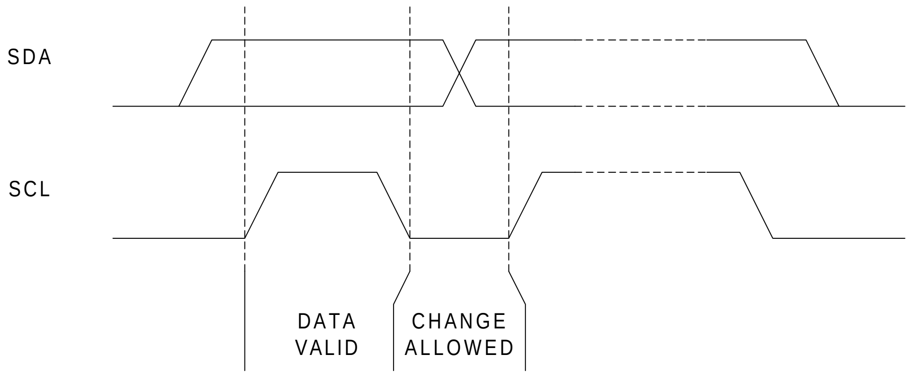


0951A-A–9/97

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. Multiple 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].

Connection

Both I^{2}C 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 3. Physical connection to the I^{2}C bus

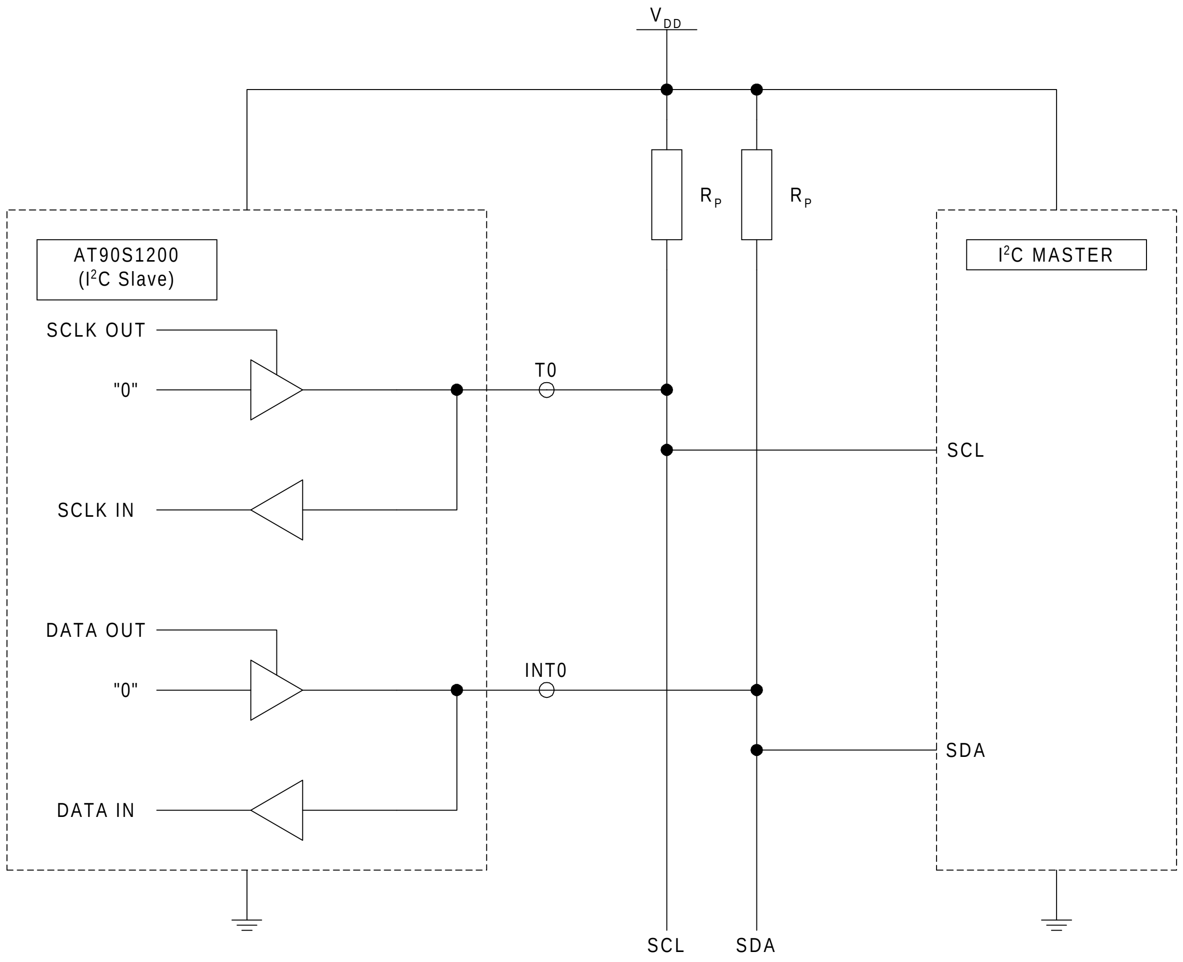
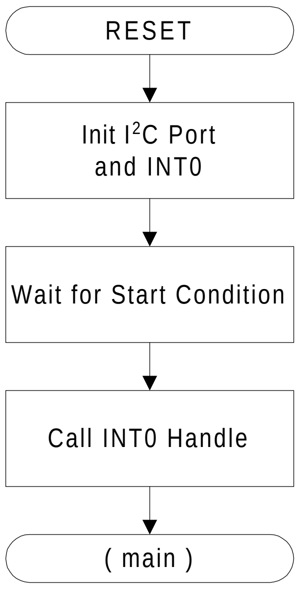


Figure 3 shows how to connect the microcontroller to the I^{2}C bus. The value of R_P depends on V_{DD} and the bus capacitance (typically 4.7k). Since SDA is connected to INT0, a falling edge on the SDA will cause an interrupt when a START condition is detected.

Implementation

The implementation of the I^{2}C slave device presented in this application note is divided into two main parts. These are a special initialization sequence executed directly after a reset and the interrupt handling routine. Flow charts is shown in Figure 4 and Figure 5.

Figure 4. Initialization Flow Chart



The initialization routine ‘i2c_init’ (Figure 4) perform the necessary initialization of PORTD and External Interrupt 0. Note that the port initialization shown in the program code really has no effect since both DDRD and PORTD registers are zero after reset. However if other pins on port D needs to be initialized, this could be done here.

When initialization is done, the routine enters into a busy-loop which waits for the first START condition. This is done because a high to low transition on SDA not necessarily indicates a START condition if the bus is not free (no activity). Hence, both SDA and SCL must be monitored.

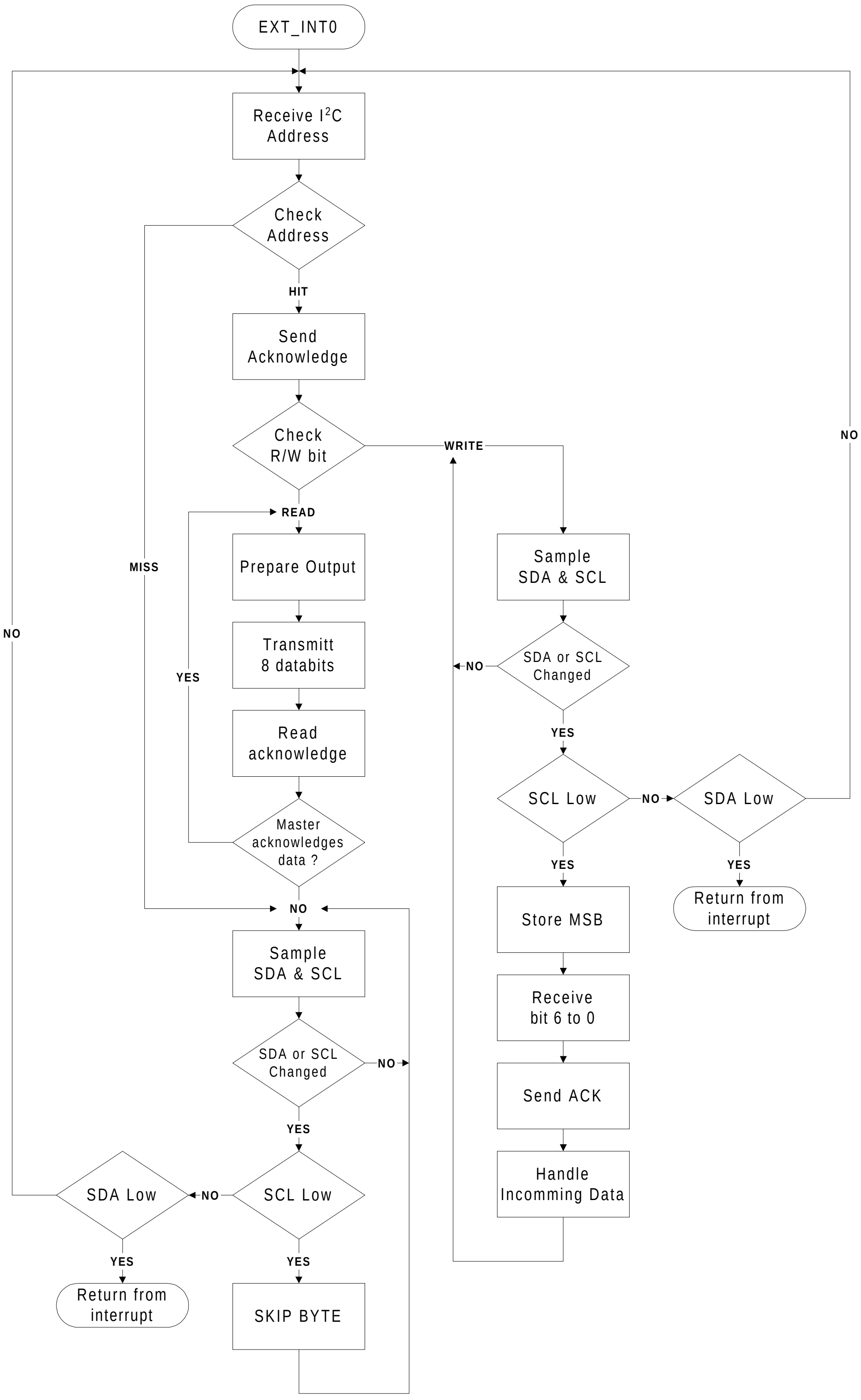
When a START condition is detected, a call to the interrupt service routine handles the first transfer and enables the interrupts.

The interrupt handling routine (flow chart shown in Figure 5) is a combination of two routines ‘i2c_wakeup’ and ‘i2c_skip’. The combining of routines keeps the code size down.

’i2c_wakeup’ detects start condition (edge interrupt on EXT_INT0) and handles the data transfer on the I^{2}C bus. There are two important locations in this routine where the user can add own code. One part handles incoming data and the other handles outgoing data. In the program code these parts are commented with **‘INSERT USER CODE HERE’**. Received data or data to be send must be placed in the ‘i2cdata’ register.

’i2c_skip’ handles situations where data transfer on the I^{2}C bus is not addressed to this device. Recall that if the bus is not free, both SDA and SCL must be monitored for a START (or STOP) condition. In this implementation timer/counter0 is used to count 8 SCL clocks i.e. one byte. By using the timer/counter 0 overflow interrupt, processing time is freed while one byte is transferred. When timer overflow occurs, the ‘i2c_skip’ is called and a new condition test is done.

Figure 5. Interrupt Handling Flow Chart



Performance Figures

Parameter	Value
Code Size	160 words
Execution cycles	N/A
Register Usage	Low registers :None High registers :5 Global :5
Peripherals Usage	2 I/O Pins, Timer/Counter0
Interrupt Usage	Timer/Counter0 Overflow Interrupt External Interrupt0

Register Usage

Only five registers are used in this implementation: ‘temp’, ‘etemp’, ‘i2cdata’, ‘i2cadr’ and ‘i2cstat’. Both temporary registers are free to be used inside the interrupt user code.

Register	Description
r16 - ‘temp’	Temporary internal register.
r17 - ‘etemp’	Temporary internal register.
r18 - ‘i2cdata’	Contains current received or transmitted data. Only valid inside interrupt user code.
r19 - ‘i2cadr’	Contains current i2c address and direction bit. Do not use for other purposes.
r20 - ‘i2cstat’	Temporary storage for SREG.

Tips and Warnings

The I^{2}C routine presented can be reduced in size if the application guarantees that the bus is free (no activity) before initialization is done. As an example, this can be done by letting all masters wait approximately 20 ms after power up before accessing the I^{2}C bus. This will ensure a free bus and eliminates the need to sample both SCL and SDA while waiting for the first START condition. The initialization will then consist of interrupt enabling, only. This procedure gives a reduction of 12 instructions.

Another size reducing method is possible by replacing the interrupt handling routines with a polling routine. However this is not recommended due to the reduction of processing time for other duties.

Handling incoming and outgoing data is time critical. The user should insert wait states if the code part which handles incoming or outgoing data is too time consuming (refer to program code for recommended sizes). For normal mode I^{2}C operation it’s also possible to increase the crystal frequency.

Procedure for insertion of wait states:

1. Right before the user code, force the SCL line low to initiate the wait state.
2. Do the user code.
3. Finish the user code by releasing the SCL line.

Program code example (inside user code):

```
sbi   DDRD,DDD4   ; force SCL low to initiate the
                  ; wait state
...               ; User data handling code
cbi   DDRD,DDD4   ; release SCL to remove the
                  ; wait state
```

Conclusion

This application note shows how to implement the AT90S1200 as a multi purpose I^{2}C peripheral device. Normal mode I^{2}C transfer (100 kHz) is supported by using a 3 MHz or faster crystal or resonator, while fast mode (400 kHz) only is supported for 16 MHz crystals. The use of interrupts to detect bus activity frees processing resources when the device is not accessed.

References

[1] *The I^{2}C-Bus and How to Use It (Including Specifications)*, Philips Semiconductors, April 1995.