# I$^2$C Slave Kernel for

## Stir in BascomAVR, add

Vladimir Mitrovic (Croatia)

**With all the processing power under the hood of the Atmel ATtiny13 and ATtiny2313 micros, it's not too difficult to get them interfaced to the I$^2$C bus: some Basic and embedded assembly code will do just nicely.**

The I$^2$C bus (a.k.a. *IIC* or *inter-IC* bus) as well as integrated circuits designed to work on it, have been described numerous times in this magazine, just look at references [1] and [2] for two recent articles. Although there exist hundreds of ICs with I$^2$C compatibility, each and every one of these will have one specific function it was designed to handle. For example, humidity sensing, relay driving, LCD controlling, data memorising, push button activity detection, keyboard decoding, bus current boosting — you mention it, it's all available, and cheap too! However, the true electronics enthusiast is creative and wants to be able to

1. glue home-brew things to the I$^2$C bus;
2. design his/her own I$^2$C peripheral(s) to do just what's desired using preferred components.

For both, it is necessary to have a minimum of I$^2$C slave functionality and then take it from there with all levels of flexibility. If this sounds like 'software', it is: *if you can program it, just do it*.

### Technical Spec

- **Atmel ATtiny13 or ATtiny2313 programmed to act as a slave device on the I2C bus**
- **Mix of BascomAVR and assembly code**
- **Software open-ended and free to community**
- **No fixed device address**
- **Learning mode and hardware activation built in**

## Project outline and hardware

The AVR microcontroller is programmed to act as an I$^2$C slave. The write address of this slave is memorised in its internal EEPROM, at the address 0, bits 7-1. Bit 0 should be zero. The most important I$^2$C rules are implemented in the program: it recognizes multiple STARTs as well as unexpected START and STOP signals in the middle of a data sequence and keeps the SCL line Low while preparing the byte to be sent to the master. The program is optimised for speed and with an 8 MHz oscillator it will accept an I$^2$C clock with a frequency of up to 400 kHz. This means that the ATtiny2313 or ATtiny13 can make do with their internal oscillators.
As everything is realised in software, no external components are needed and we arrive at bare-bones circuit diagrams shown in **Figure 1** and **Figure 2**. The pull-up resistors on the SCL and SDA lines may be omitted if they are provided elsewhere on the I$^2$C bus. For hardware, it doesn't get simpler than that.

## The kernel

The program has several time-critical routines that are written in assembler. These routines constantly monitor the traffic on the I$^2$C bus and call the appropriate subroutines if a valid write or read address is recognized. The program kernel acts as any other I$^2$C slave, with the following conditioned functionality.

**1.** If the first byte following the START signal is recognized as its own write address (bit 0 = 0), it will:
- confirm it with ACK;
- accept up to two following bytes, confirming each of them with ACKs;
wait for RESET (any additional byte before RESET is ignored);
- call the 'Process_received_data' subroutine after RESET;
- wait for the next START.

**2.** If the first byte following the START signal is recognized as its own read address (bit 0 = 1), it will:
- confirm it with ACK;
- pull down the SCL line to signal the I$^2$C master that data is prepared (delayed SCL);
- call the 'Prepare_data_for_master' subroutine;
- free the SCL line to enable further communication;
- send one byte of data contained in the 'Data_for_master' variable to the master;
- wait for the next START.

**3.** If the first byte following the START signal is not recognised as its own write or read address, it will ignore all communication on the I$^2$C bus until the new START signal.

It is the programmer's (i.e. your!) responsibility — and challenge — to provide the code for the 'Process_received_data' and 'Prepare_data_for_master' subroutines in order to process received data or to prepare data to be sent to the host. In order to facilitate programming even for a not very skilled programmer, the assembler ker-

# ATtiny13 and '2313
## a pinch of assembly code

nel is embedded in the BascomAVR structure. So, the programmer may use BascomAVR from MCS Electronics [3] (even the demo version) to adjust the program to his/her own needs.

## Examples

Two examples are provided to get cracking: 'I2C_slave_ATtiny13_Elektor.bas' and 'I2C_slave_ATtiny2313_Elektor.bas'. The examples differ only in details specific to the microcontroller used: ATtiny13 or ATtiny2313. The kernel is placed before the 'End' statement, and the user subroutines and any data for the user to adjust, after the 'End' statement.
In these examples the microcontrollers act like an AT24C0x serial EEPROM with its internal address set to $EA (write) and $EB (read). Write address $EA should be programmed in the first location of EEPROM (address 0). The BascomAVR compiler will produce an '.eep' file for the programmer, like this:

```
$eeprom
Data &HAE    ,address
of this I2C slave
$data
```

The 'Process_received_data' subroutine is called after the STOP signal if a valid write address of this I$^2$C slave is recognised before. Up to two bytes of data following the address are memorised as shown in **Table 1**.
In this example, I2C_b1 is used as the address of the internal EEPROM and I2C_b2 as data to be written — be careful not to rewrite address 0!

```
Process_received_data:
   Writeeeprom I2c_b2 , I2c_
b1  'I2C_b2 byte is written to
'internal eeprom at I2C_b1
   Waitms 5  'wait until written
Return
```

The 'Prepare_data_for_master' subroutine is called immediately after the valid read address of this I$^2$C slave is recognised. No data are passed to the subroutine, but the subroutine should prepare one byte of data to be sent to the master in the variable 'Data_for_master'. Keep in mind, however, that preparing data for the master should be as quick as possible because the master usually does not wait for data indefinitely.
In this example, one byte is read from the internal EEPROM from the address in I2C_b1 (received before, according to the communication protocol for AT24C0x):

```
Prepare_data_for_master:
    Readeeprom Data_for_master ,
I2c_b1   '1B is read from internal
eeprom at I2C_b1
Return
```

The examples were tested on the circuits in Figures 1 and 2 and have worked well with an SCL of up to 400 kHz. Bear in mind that the ATtiny13 has only 64 bytes of internal EEPROM, while ATtiny2313 has 128 bytes, so theoretically the ATtiny2313 could replace an AT24C01. However, the real advantage of this program is that the microcontroller may play the role of any 'new' I$^2$C slave with specific address, according to one's needs. For example, it may be used as an interface between the I$^2$C master and any equipment.

## Learning mode

It was explained before that the I$^2$C address of this I$^2$C slave is memorised as the first byte in the internal EEP-

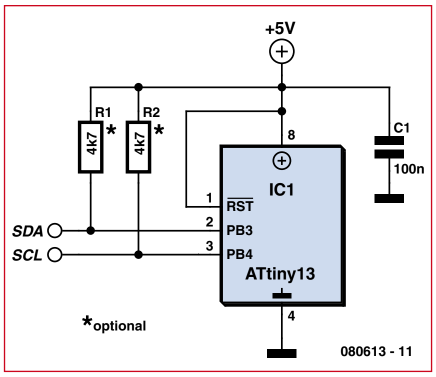

Figure 1. ATtiny13 as an I$^2$C slave.

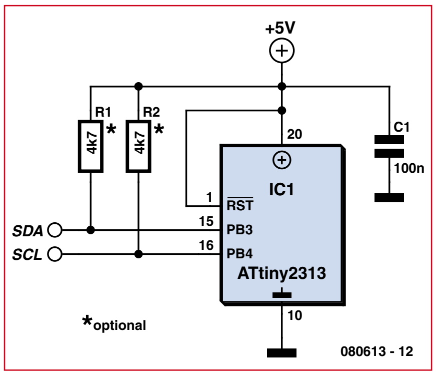

Figure 2. ATtiny2313 as I$^2$C slave.

**Table 1. Data byte memory**

| variable name | variable type | description |
|---|---|---|
| I2C_b1 | Byte | 1st byte after I$^2$C address, = 0 if not received (i.e. if STOP signal occurred first) |
| I2C_b2 | Byte | 2nd byte after I$^2$C address, = 0 if not received (i.e. if STOP signal occurred first) |
| I2C_stop | Byte | = 255 if STOP signal received, = 0 if not (only for debugging purposes, should always be = 255) |

ROM. It may be defined in the program and programmed in EEPROM after programming the flash memory as explained in the example. As an additional benefit, the microcontroller may be set in 'learning mode' and accept and memorise the new address from the I²C bus. There are two ways to activate the learning mode, software and hardware. In both cases, the 'Learning_mode' flag is set and the first I²C address (1st byte after the next START signal) will be accepted and memorized as the new I²C address of this slave. Immediately after this procedure the 'Learning_mode' flag is erased and the microcontroller acts as explained before. The whole procedure is already programmed in the kernel; you only have to set the flag!

An example of software activation of the learning mode is given below. Here, we check if a specific data combination is received and set the 'Learning_mode' flag if it is:

```
Process_received_data:
   ...
   ...
   If I2c_b1 = xxx and I2c_b2
= yyy Then  'check if specific
key combination is received
      Learning_mode = 1
'set learning flag
   End If
   ...
   ...
Return
```

## hardware activation

Use any free I/O pin, e.g. PINB.0 and provide a jumper or pushbutton to connect it to the GND. When you want to reprogram the I²C address of this slave, keep the pushbutton pressed for two seconds at power-up or reset. At the beginning of the program the 'Learning_mode_hw_activation' subroutine is called. This subroutine is normally empty (i.e. it contains no more than 'Return'). If you want to enable hardware activation of the learning mode, you should check the state of the chosen I/O pin (Pinb.0 in this example) and set the Learning_mode flag if the expected condition occurs (Pinb.0 = 0 in this example):

```
Learning_mode_hw_activation:
   Config Pinb.0 = Output
   If Pinb.0 = 0 Then
      Learning_mode = 1
   End If
Return
```

## Resources

The Basic programs (written in BascomAVR) and hex object code files for the microcontrollers are available as a free download from the Elektor website [4]. Due care should be taken with the hex files as they may not be compatible with every programming system for the ATtiny13 and '2313 microcontrollers. In case of doubt, compile the .bas files locally and work out the compatibility with your programmer.

(080613-I)

## References and Internet Links

[1] The Secrets of I2C, Elektor March 2008.

[2] Bits on Parade, Elektor December 2008.

[3] www.mcselec.com

[4] www.elektor.com/080613