CIRCUIT CELLAR ONLINE

FEATURE ARTICLE

Dariusz Caban

Software Implementation of the I^{2}C Protocol

Thanks to the possibility of all-software implementation of the I2C protocol, microcontrollers can communicate with I2C devices. With this article, Dariusz presents us with an example implementation of the Standard mode of the I2C protocol for the popular 8031 microcontroller.

External devices that are used to expand a microcontroller's internal resources are generally available with a parallel interface but a serial interface is becoming more popular. There is a wide range of such devices, including EEPROMs, display controllers, real-time clocks, A/D and D/A converters, and I/O expanders. Serial thermal sensors, directly connectable to a microcontroller, are offered as well. Serially accessed devices require less wiring and space on printed circuit boards than parallel ones. Thus, printed circuit boards and connectors can be simpler and smaller. Reduced wiring also increases reliability of a system. In some cases, such microcontrollers without an external address and data bus, the application of serial devices enables interaction between the microcontroller and the outside world.

Most serial devices are equipped with a synchronous interface for which they require separate signal lines for transferring data and clock information. The clock, called the master, must be supplied by a microcontroller. This clock frequency does not have to be constant. The master also initiates communication with serial devices called slaves. Often, multiple slaves can use common data and clock lines, with each slave having its own select line or address.

One of the most popular synchronous interfaces is the inter-integrated circuit (I^{2}C), which was developed by Philips nearly 20 years ago. I^{2}C is a low-bandwidth, short-distance, two-wire interface that was originally designed to enable communication between devices inside a TV. Now, I^{2}C interface is implemented in over 1000 different devices. [1] Some microcontrollers have a hardware I^{2}C controller built-in (e.g., the P8xC528 from Philips and the PIC16C6*x* from Microchip). An all-software implementation of I^{2}C protocol is also possible. Thanks to this method, any microcontroller can communicate with I^{2}C devices. In this article, I'd like to present such an implementation for the popular 8031 microcontroller.

THE BUS

The I^{2}C interface is modest in its hardware resource requirements, because only a single pair of signal lines is needed: serial data (SDA) and serial clock (SCL) (see Figure 1). Both lines are bidirectional and must be connected to a positive supply voltage via pull-up resistors. The SDA and SCL pins of each device also must have an open drain or open collector in order to perform the wired AND function. Data can be transferred at a rate of up to 100 kbps in Standard mode, up to 400 kbps in Fast mode, and up to 3.4 Mbps in High-Speed mode. Each slave on the bus is identified by a unique address.

In Standard mode, 7-bit addressing is used. In other modes, slaves can have 7- or 10-bit addresses. The num-

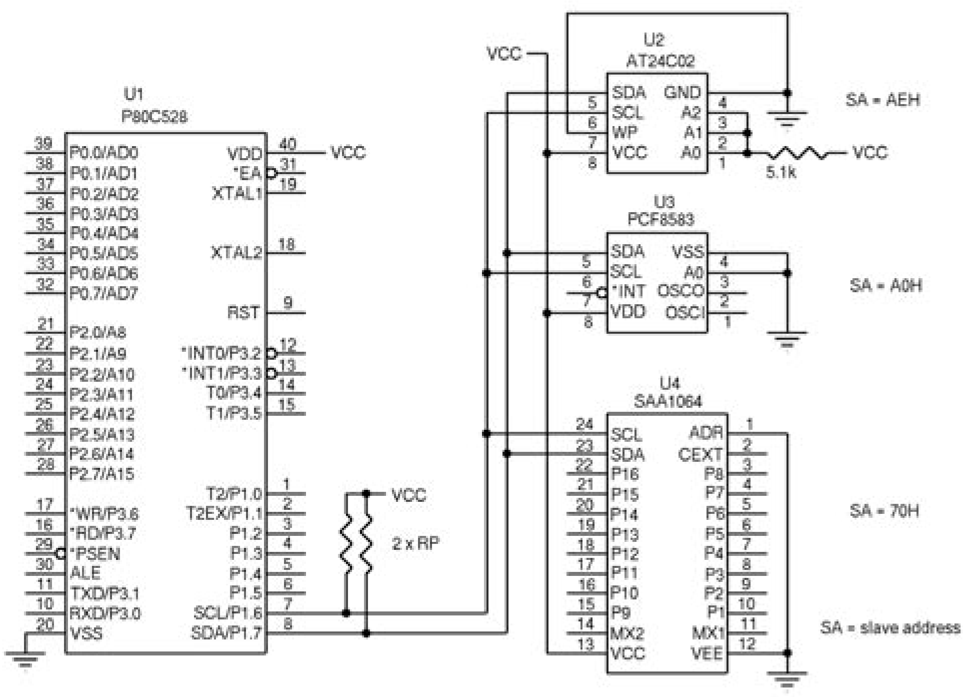

Figure 1—*Here you can see how the I²C devices connect to the microcontroller.*

ber of devices that can be connected to the same bus is limited by the maximum bus capacitance of 400 pF.

The I²C bus can be controlled by more than one master. If two or more masters simultaneously initiate data transfer, collision is detected and an arbitration procedure is performed. The arbitration doesn't cause data corruption, however, most system designs include only one master.

Only the master generates the clock, but transmission speed can be adjusted to the internal operating rate of the addressed slave. This adjustment is made by clock stretching, in which the slave keeps the SCL pulled low until it is ready to continue.

THE PROTOCOL

The I²C protocol is level-sensitive. The data must be stable when SCL is high. Except for two situations, the state of the SDA line can only change when SCL is low. The exceptions have special meanings (see Figure 2). A 1-to-0 transition signals the beginning of a transfer and is termed as a start condition. A 0-to-1 transition signals the end of a transfer and is termed as a stop condition. The data is transferred in bytes, with the most significant bit sent first. Note that the byte transfer requires nine clock pulses. The transfer of a byte's bits takes eight pulses, and the ninth is used for acknowledgment. Between start and stop conditions, an unrestricted number of bytes can be transferred.

After a start condition, the byte containing the slave address (or part of the address, when 10-bit addressing is used) and a data direction bit is always sent first (see Figure 3). A start condition can be repeated without first generating a stop condition. This is used to change transfer direction or to address another slave. If there is no acknowledgment from the addressed slave (because it is not connected or performs some internal operation), the master can abort the transfer. Next, if the slave is being written to, it must acknowledge each byte received. Lack of acknowledgment indicates that it cannot accept data. While reading from the slave, the master is also obliged to acknowledge each byte, except the last byte.

The master can communicate with the slave according to several scenarios called transfer formats. For example, there are three possible formats when 7-bit addressing is used. Their descriptions can be found in "The I²C-Bus Specification—Version 2.1," by Philips. [1]

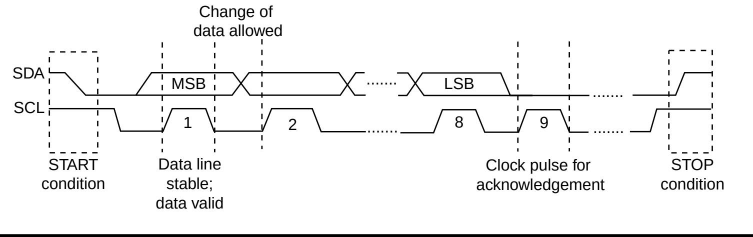

Figure 2—*This diagram shows the issuing of the start and stop conditions and how the byte transfer on the I²C bus is performed.*

THE PROCESS

The I²C protocol does not have to be implemented in hardware. Software implementation is also possible, because the protocol is forgiving with regard to timing accuracy. And, any of the microcontroller's general-purpose I/O lines can be used as I²C lines. This approach is useful when a system design includes only a single master.

As stated earlier, such designs are most frequent. I have implemented the I²C protocol for the 8031 microcontroller, using only Standard mode. The source code was written in C-51. The use of a high-level programming language shortened development time considerably. It also simplifies changes and adaptation of the code to microcontrollers with different architectures.

Listing 1 presents functions performing basic operations of the I²C protocol. The functions given require 135 bytes of code memory and only a few bytes of internal data memory, if a compact memory model is set. In order to guarantee appropriate timing characteristics of signals, NOP instructions are used. An 8031 microcontroller executes a NOP instruction in one machine cycle. [2] One cycle takes $12/f_{osc}$ s, where f_{osc} is the oscillator frequency. The number of NOP instructions in the given functions was selected based on an oscillator frequency of 12 MHz.

Now, let's use the set of functions given in Listing 1 to implement operations on the Atmel AT24C02 device you saw in Figure 1. This device supports 256 bytes of EEPROM. The slave address of the AT24C02 consists of a 4-bit type identifier (1010), followed by a 3-bit sequence, which corresponds to logic levels on the A2, A1, and A0 inputs. This way, up to eight EEPROMs can be addressed on the same I^{2}C bus. The AT24C02 also has a write protect (WP) pin that provides hardware data protection.

The following operations are allowed: byte write, page write, acknowledge polling, current address read, random read, and sequential read. Listings 2 and 3 present example functions performing byte write and sequential read operations, respectively. It was assumed that a single EEPROM exists in the system.

When the microcontroller terminates the write sequence with a stop condition, the EEPROM enters an internally timed write cycle that, for the AT24C02, can last up to 10 ms. During the write cycle, the EEPROM is busy and ignores all communications on the I^{2}C bus. The ready/busy status of the device is determined by using an acknowledge polling operation. This operation involves issuing of a start condition followed by the slave address byte. If the EEPROM does not acknowledge, the cycle is still in progress. If it does, the cycle has completed. Listing 4 presents the function performing the acknowledgement of the polling operation and its possible use.

The sequential read operation is a convenient way to get multi-byte values stored in the EEPROM. In the function presented in Listing 3, this operation is initiated by a write sequence to load the EEPROM's inter-

Listing 1—*By using these functions, data transfer via an I^{2}C bus can be performed, assuming only a single master exists.*

```
#define uchar unsigned char

#define SDA P1.0      /* microcontroller's I/O lines */
#define SCL P1.1      /* assigned to I2C lines       */

/****************************************************
  Issuing of START condition.
 ****************************************************/

void start(void)
{
  SDA = SCL = 1;
  SDA = 0;
  _opc(0);           /* it places NOP instruction */
  _opc(0);           /* into executable code      */
  _opc(0);
  _opc(0);
  _opc(0);
  SCL = 0;
}

/****************************************************
  Issuing of STOP condition.
 ****************************************************/

void stop(void)
{
  SDA = 0;
  SCL = 1;
  _opc(0);
  _opc(0);
  _opc(0);
  _opc(0);
  _opc(0);
  SDA = 1;
}

/****************************************************
  Clock pulse generation. The function returns data
  or acknowledgment bit.
 ****************************************************/

bit clock(void)
{
bit level;           /* state of SDA line         */

  SCL = 1;
  _opc(0);
  while(!SCL);       /* if a pulse was stretched */
  _opc(0);
  _opc(0);
  _opc(0);
  level = SDA;
  _opc(0);
  _opc(0);
  SCL = 0;
  return(level);
}

/****************************************************
  Writing a byte to a slave, with most significant
  bit first. The function returns acknowledgment bit.
 ****************************************************/

bit write(uchar byte)
{
```

Continued

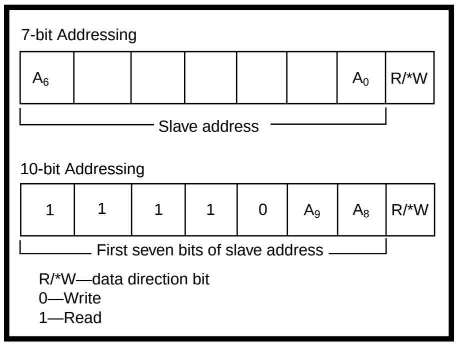

Figure 3—*The byte containing all or part of the slave address and the data direction bit are sent first after a start condition.*

nal address counter with the initial value. Then the master issues a start condition again, sends a slave address with the data direction bit high, and begins reading. After the master receives a byte and acknowledges it, the EEPROM increments the address counter and sends a successive byte. The EEPROM continues sending until the master does not acknowledge and generates a stop condition. Listing 5 shows an example of using the *EEPROM_sequential_read()* function.

CONCLUSIONS

Compared to other competing synchronous serial interfaces, Microwire from National Semiconductor and SPI from Motorola, I²C has the least hardware requirements. Only two I/O pins of the microcontroller are needed to communicate with multiple slaves, because each slave is identified by its unique address, not by a separate select line. Also, because the I²C protocol is level-sensitive, its noise immunity is likely to be higher than in edge-sensitive competitors. And, unlike Microwire and SPI, I²C slaves provide feedback to the master, which indicates whether or not transmission was successful. Until recently, the I²C protocol was significantly slower, but in 1999 a high-speed mode was introduced, which offered rates up to 3.4 Mbps.

In this article, I have presented an example all-software implementation of the Standard mode of the I²C protocol. The source code was written in a high-level language, and you can easily see that it is not complicated. Although the compiler used was rather old [2], small-size executable code was produced.

Dariusz Caban completed his studies at Silesian Technical University in Gliwice, Poland, where he received his MS. Since then, he has been working at the Institute of Theoretical and Applied Computer Science, Polish Academy of Sciences (IITiS PAN) and cooperates with manufacturers of measurement and control equipment. He holds a Ph.D. and specializes in programming of microcontrollers, mainly in high-level languages. You may reach him at darek1@mail.iitis.gliwice.pl.

REFERNCES

[1] Philips Semiconductors, "The I²C-Bus Specification—Version 2.1," January 2000, http://www.semiconductors.philips.com/acrobat/various/i2c_bus_specification_3.pdf.
[2] Intel Co., "Embedded Controller Handbook," 1987.

Listing 1— *continued*

```
uchar mask = 0x80;

  while(mask)
  {
    if (byte & mask)
      SDA = 1;
    else
      SDA = 0;
    clock();
    mask >>= 1;      /* next bit to send           */
  }
  SDA = 1;           /* releasing of the line      */
  return(clock());   /* a slave should acknowledge */
}

/***************************************************
  Reading byte from a slave, with most significant
  bit first. The parameter indicates, whether to
  acknowledge (1) or not (0).
 ***************************************************/

uchar read(bit acknowledgment)
{
uchar mask = 0x80,
      byte = 0x00;

  while(mask)
  {
    if (clock())
      byte |= mask;
    mask >>= 1;      /* next bit to receive */
  }
  if (acknowledgment)
  {
    SDA = 0;
    clock();
    SDA = 1;
  }
  else
  {
    SDA = 1;
    clock();
  }
  return(byte);
}
```

SOFTWARE

The source code is available for download in the html format of the article.

SOURCES

C-51
Archimedes
+38121618243
Fax: +38121619308
www.archimedes.co.yu

AT24C02 Two-wire serial EEPROM, AT89Cx051 80C31 microcontroller
Atmel Corp.
(408) 441-0311
www.atmel.com

PIC16C6*x* 8-bit CMOS Microcontroller
Microchip Technology Inc.
(978) 692-3848
Fax: (978) 692-3821
www.microchip.com

P80C528, P83C528 8-bit Microcontrollers
Philips Semiconductors
(800) 326-6586
(212) 536-0500
Fax: (212) 536-0559
www.semiconductors.philips.com

Listing 2—*This function writes bytes to the EEPROM cell located at a given address. The function returns the status of operation.*

```
#define EEPROM 0xAE        /* slave address, data direction
                              bit = 0 */

bit EEPROM_byte_write(uchar address, uchar byte)
{
bit status;

  status = 0;               /* failure by default */
  start();
  if (!write(EEPROM))      /* write operation    */
    if (!write(address))   /* byte address       */
      if (!write(byte))
        status = 1;         /* success            */
  stop();
  return(status);
}
```

Listing 3—*The bytes are placed in memory, starting from the block address.*

```
#define NO_ACK 0
#define ACK    1

bit EEPROM_sequential_read(uchar *block, uchar address,
                           uchar size)
{
bit status;

  status = 0;                      /* failure by default */
  start();
  if (!write(EEPROM))             /* write operation    */
    if (!write(address))          /* initial address    */
    {
      start();
      if (!write(EEPROM | 0x01)) /* read operation     */
      {
        while(size--)
          *block++ = read(size ? ACK : NO_ACK);
        status = 1;                /* success            */
      }
    }
  stop();
  return(status);
}
```

Listing 4—*The first function determines the ready/busy status of the EEPROM. If you assume that after the write sequence a program execution may be suspended until the write cycle is complete, the EEPROM_busy() function can be used.*

```
bit EEPROM_acknowledge_polling(void)
{
bit status;

  start();
  status = write(EEPROM);
  stop();
  return(status);   /* if 1, the write cycle is in progress */
}

void EEPROM_busy(void)
{
  while(EEPROM_acknowledge_polling())
    delay(1,164);   /* about 1 msec */
}
```

Listing 5—*This example shows how to restore float variables within the contents of the EEPROM.*

```
#define T1_SETP_ADDR 0   /* starting locations of set point */
#define T2_SETP_ADDR 4   /* values of temperatures           */

                           :

/****************************************************
  Global variables.
 ****************************************************/

float t1_setp,          /* temperatures' set points */
      t2_setp;

                           :

/****************************************************
  Initialization of the system.
 ****************************************************/

void initialization(void)
{
                           :
  EEPROM_sequential_read(&t1_setp,T1_SETP_ADDR,4);
  EEPROM_sequential_read(&t2_setp,T2_SETP_ADDR,4);
                           :
}
```