Firmware Emulation of an I2C Slave Device

by Steve Kolokowsky, Cypress Semiconductor

Many embedded microprocessors contain a two-wire serial interface, which looks remarkably similar to Philips I²C (I2C) interface. In many designs these two pins are an unused access point into the microprocessor. This article will provide a detailed look at how to implement the slave side of the I2C interface using bit banging. Soon you will be able to use two GPIOs on your own processor for inter-processor communication, which will save a lot of pins over an 8-bit parallel bus.

The I2C standard is covered by patents owned by the Philips Semiconductor spin-off, NXP, created in 2006. In October 2000, Philips was aggressively enforcing these patents, filing a lawsuit against six semiconductor companies. However, according to NXP's website, "I²C licenses under the remaining patents in the program will be free of any royalties, for any use of the patents after October 1, 2006, without any prejudice to any claims for past use whatsoever. Also the fee for obtaining an I²C slave address allocation will not be affected and will continue to apply until further notice."

I2C Overview

The I2C interface is a shared two-wire serial bus. The bus allows communication between a single master and a single slave at any given time, but multiple masters and slaves can share the same bus. CPU devices are typically masters and peripheral devices are usually slaves. The master device initiates the transaction and provides the clock. The slave devices are identified by a 7-bit address, theoretically allowing up to 127 slave devices on any I2C bus. Fig. 1 shows the I2C bus topology.

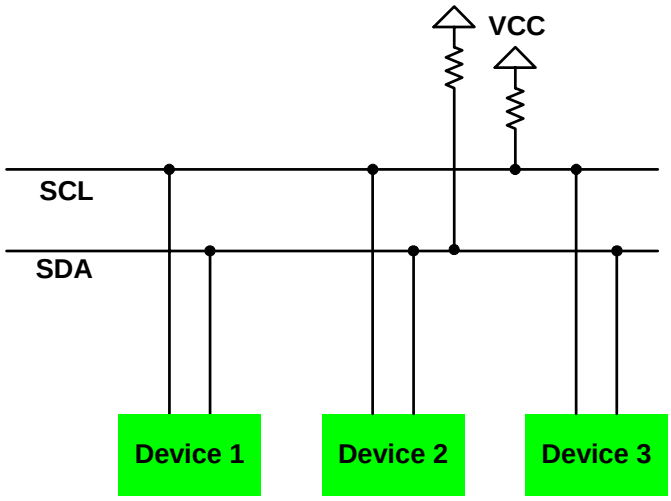

Fig. 1: I2C Bus Topology

Each bus transaction is initiated by the master device. The first byte of the transaction is a 7-bit address field and a single read/write (r/w) bit. If a slave device matches the address, it responds to the address byte with an ACK bit by pulling SDA low during the 9th bit time. If the r/w bit is 1 (read), the slave device sends one or more data bytes to the host which the host ACKs in the same fashion. The slave stops sending data when it receives a STOP bus condition from the host. If the r/w bit is 0 (write) the roles are reversed with the master sending data and the slave sending the ACK.

I2C defines two special bus conditions called START and STOP see Fig. 2). START occurs when there is a falling edge on SDA with SCL held high. After the START condition, the master drives SCL low, and then sends the address byte. A STOP condition is the opposite of START, a rising edge on SDA with SCL high. During normal DATA and ACK transmission, the data bus is stable when SCL is high, transitioning when SCL is low. The bus is idle when both lines are 1 (VCC).

SCL and SDA are both wired-or signals implemented with open-collector/open-drain I/O cells. Open collector signals are driven low, but not driven high. The bus is pulled high by a resistor attached to VCCIO. This allows multiple devices to share the bus at the same time and minimizes concerns about transitions between transmission and reception. The use of wired-or signals even allows multiple master devices to collide on the bus without damaging each other's drivers.

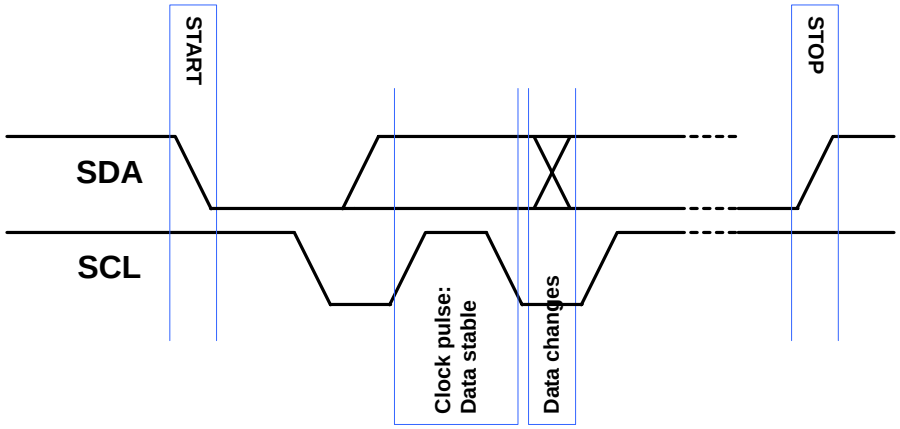

Fig. 2: I2C START And STOP Conditions

I2C Emulation In Firmware

Emulating I2C via GPIOs requires attention to four protocol layers:

- Hardware interface: simulating open drain i/o cells
- Bit-level protocol: START, STOP, data
- Byte-level protocol: address, data, ACK
- Application-level: EEPROM, temp sensor, tuner, etc

To understand the sample code below, the following definitions must be understood:

- SCL, SDA: bit-wide variables mapped to the I/O ports controlling SCL, SDA pins. The same address is used for input and output values
- OEA: output-enable register containing both SCL and SDA bits
- bmSCL, bmSDA: bit masks for SCL, SDA bits within OEA

The hardware-interface layer is an easy firmware task. To truly emulate an open-collector I/O cell, only enable the driver when the signal is low.

The primitives defined in Fig. 3 will be used by the higher-layer code below.

```
Init()
  { OEA &= ~bmSDA; OEA &= ~bmSCL;  // float the i/os
        SDA = 0; SCL = 0;}                  // Drive 0 when on
Drive1()
  { OEA &= ~bmSDA; }                        // Allow pullup to work
Drive0()
  { OEA |= bmSDA; }                         // Drive to 0
```

Fig. 3: Bit-Level Functions

On a slave device, the bit-level protocol is the hardest part of I2C to implement because of the strict timing requirements in the spec. In a 100-kHz device the master can begin sending SCL pulses 4 µs after START. In a 400-kHz device this limit is just 600 ns! Another option is to engineer your ISR so that it can operate with a starting point in the *acceptable* region shown below. This adds one SCL low time to your latency for a total of 8.7 µs at 100 kHz. Once you have seen the START, you can either stay in the ISR to process the data, or trigger another ISR on the rising edge of SCL. This ISR will also have to be very low latency, since the data are only guaranteed to be stable when SCL is high. This tHIGH parameter is 4 µs at 100 kHz, but only 600 ns at 400 kHz. In this example implementation, we remain in a loop watching SCL and SDA until STOP.

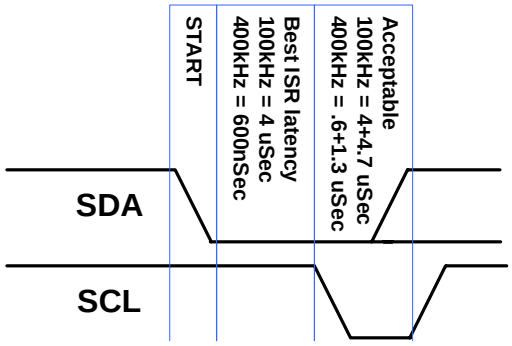

Fig. 4: START Timing

If your device cannot always meet these requirements and you can change the master device, there are two options in the spec to allow for longer timing. One is for the host to transmit a dummy address (0000 0001) called a *start byte* to allow the ISR to start. This

start byte is not used as an address by any device. This increases the allowable latency to a full START+BYTE+ACK time, almost 10 SCL clocks. Another option is to build retransmission capability into the master driver. If the slave device fails to ACK its address the first time, simply issue another START and transmit the address again.

```
void ISR_INT1_SDA(void) interrupt
{
   WORD retval;
   Bit readmode;

   // make sure this is an I2C START
   // (SDA goes low while SCL is high) and set the flag
   if (SCL)
   {
      disable_SDA_interrupt();
      retval = I2C_START_DETECTED;
   }
   else // False trigger, exit the ISR.
      return;

   // Loop for one or several START/address conditions before
   // a STOP.
   while(retval == I2C_START_DETECTED)
      {
      retval = readI2CSlaveByte(1);  // read address, 1=addr match
      if ((retval & 0xfffe) != I2C_SLAVE_ADDR)
         break;  // Other addr or STOP received, return.

      // LSBit of the returned address is the read/!write bit
      // Enter data transfer phase until STOP or START received
      if (retval & 1)
         retval = sendData();
      else
         retval = receiveData();
     }

   enable_SDA_interrupt();
   return;
}
```

Fig. 5: SDA ISR Detects START, Then Reads Or Sends Data As Needed

Once a START has been detected, the next task is to read the address sent by the master and compare it to our slave address. This is done by a generic read routine that can read addresses or data. The only difference between reading an address and reading data is that the slave device should only ACK its own address, while it should ACK all data it

receives. This routine returns a WORD rather than a BYTE to allow special conditions (STOP, START) to be returned.

The readI2CSlaveByte() routine shown in Fig. 6 waits for the clock to become high, signaling that the data line is stable and can be read. The data bit is stored, and the routine waits for another low to high transition on SCL until 8 bits have been read. While SCL is high, SDA is monitored for START or STOP conditions.

```
WORD readI2CSlaveByte(bit addrMatch)
{
  // Read 8 bits from I2C (MSB-->LSB), respond with ACK
  // SCL could be high or low coming in here, depending on CPU speed
  BYTE i;
  BYTE retval = 0;

  for (i = 0; i < 8; i++)
  {
    while (!SCL) ;
    retval = (retval << 1) | SDA;
    while (SCL)
    {
      // If SDA changes while SCL is high, it's a
      // stop (low to high) or start (high to low) condition.
      if ((retval & 1) != SDA)
        return((SDA) ? (I2C_STOP_DETECTED) : (I2C_START_DETECTED));
    }
  }
  // Send ACK -- SCL is low now
  // 100Khz hold time spec is 0-3.45usec, high speed is 0-.9usec
  if (!addrMatch || ((retval & 0xfe) == I2C_SLAVE_ADDR))
  {
    Drive0();
    while (!SCL) ;
    while (SCL);
    Drive1();
  }
  else
  {
    return(I2C_ADDR_MISMATCH);
  }

  return retval;
}
```

Fig. 6: readI2CSlaveByte Reads Addresses Or Data

Fig. 7 shows the high-level slave state machine. It may look daunting with all of the potential state transitions, but many of them are the same. Any START condition puts the device in *receive address* mode. Any STOP condition triggers *wait for START*. All of the other paths deal with address interpretation: An address mismatch triggers *wait for START*, since every address byte is preceded by a START. An address match with the READ bit set moves the state machine to *send data*, while the WRITE bit causes *receive data*. The ISR shown in Fig. 5 is triggered by a potential START and contains states 1, 2, and 3.

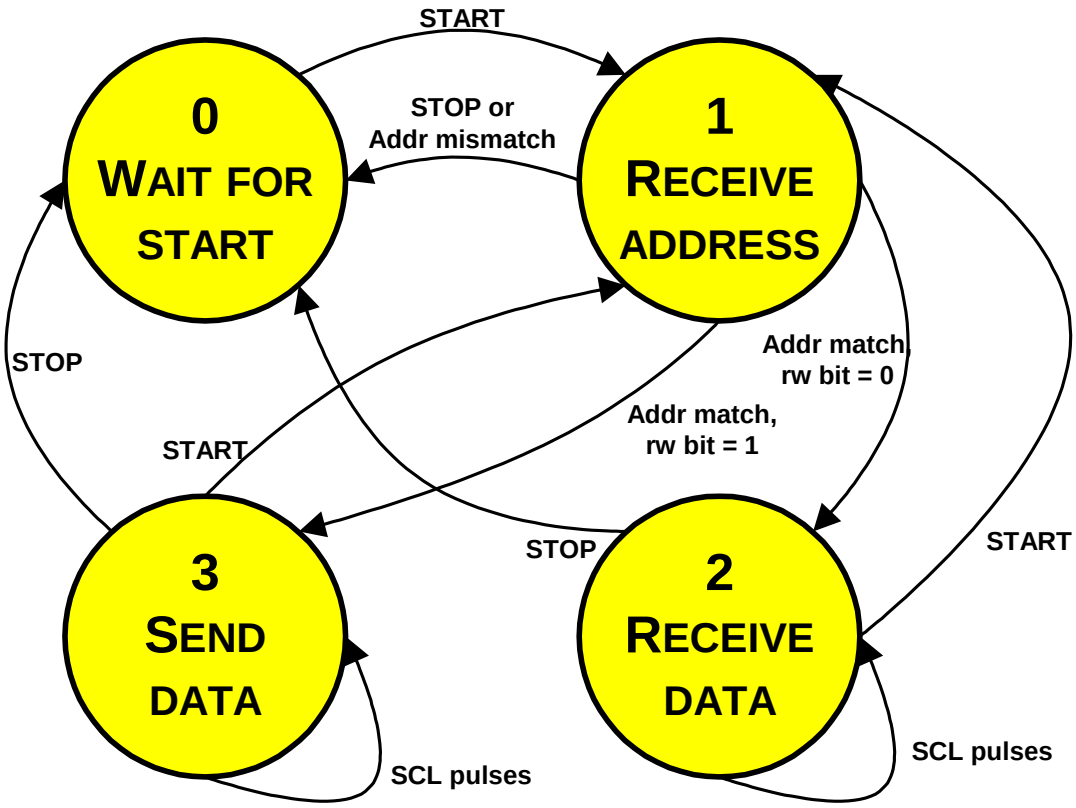

Fig. 7: I2C Slave State Machine

Although it is difficult to meet the timing requirements of an I2C slave device, it is possible with the low interrupt latencies that can be achieved without OS overhead. If both sides of the link are under your control, many options exist that can make implementation easier. Once you have an I2C slave on your device, you can free up your big 16-bit bus for more important tasks.

References

The I2C Bus Specification, version 2.1, January 2000: http://www.nxp.com

About The Author

Steve Kolokowsky is a Member of the Technical Staff in Cypress Semiconductor's Consumer and Computation Division (CCD). Steve's focus is high-speed USB peripheral products. He has participated in the design of many of today's highest selling MP3 players. Steve earned a BS in Systems Engineering from Rensselaer Polytechnic Institute in Troy, NY. He is based in sunny San Diego, California, USA. He can be reached at syk@cypress.com