CHAPTER 11

The I^{2}C Bus

Philips developed the I^{2}C (Inter-IC) bus in the early 1980s for mass-produced items such as televisions and audio equipment. The I^{2}C bus is a bidirectional, two-wire serial bus that provides a communication link between multiple integrated circuits (ICs) in a system. I^{2}C has become a generally accepted industry standard for embedded applications and has been adopted by many IC manufacturers. All devices that are compatible with the I^{2}C bus include an on-chip interface that allows them to communicate directly with each other on the bus. The I^{2}C bus supports three data transfer speeds: standard, fast-mode, and high-speed mode. All data transfer speeds modes are backward compatible. Each device on the I^{2}C bus has a unique address and can operate as either a transmitter or receiver depending on its function.

What Is the I^{2}C bus?

I^{2}C, or Inter Integrated Circuit, is a bus protocol developed by Philips Semiconductor for communication between integrated circuits. A document describing the complete bus protocol standard is freely available for download from the Philips website[1], but the protocol itself is patented and the integrated circuits that make use of it are subject to a sort of licensing agreement with Philips. I^{2}C is primarily used in embedded applications where a microcontroller communicates with and controls a variety of peripheral devices. Digital potentiometers, EEPROMS, A/D converters, phase-locked loop synthesizers, microcontrollers and audio/video products are all good examples of products that use I^{2}C. Some of the most salient features of the bus are found in the following list:

- The I^{2}C bus consists of two signals: the serial clock (SCL) and a serial data line (SDA).
- The bus is bidirectional and makes use of pull-up resistors. I^{2}C devices either pull the bus to logic low, or allow the bus pull-up to pull it high.

[1] Philips Semiconductors I^{2}C-bus – http://www-us.semiconductors.philips.com/i2c/

- The bus has three speed modes, a standard mode (<100 kHz), a fast mode (100 kHz – 400 kHz) and a high-speed mode (400 kHz – 3.4 MHz).
- Data transfer is based on 8-bit words.
- Every device on the bus has a unique address, which is either 7 bits or 10 bits wide.
- The bus is based on a master/slave device relationship. Devices can be one or the other or switch back and forth. The bus can have more than one master and features a process called "arbitration" to resolve conflicts when multiple devices try to control the bus at once.
- The number of devices on the I^{2}C bus is limited by the capacitance of the bus, which must be less than 400 pF.

The I^{2}C Bus in More Detail

We've outlined the general characteristics of the I^{2}C bus, so let's take a look at how it all works in more detail.

The master/slave concept in I^{2}C

In the Dallas Semiconductor 1-Wire bus protocol, there can be only one master on the bus, and every other device is a slave. In I^{2}C, things are more complicated. Devices can be masters sometimes, slaves sometimes, and sometimes there can be multiple masters trying to control the bus at the same time. The best way to delve into this is to introduce some basic terminology from the I^{2}C specification. It is important to note that these definitions relate to the I^{2}C bus, and if we use the same words in the context of discussing a different bus protocol, they have slightly different meanings.

Table 11-1: Definition of some basic I^{2}C terms

Term	Description
Transmitter	The device which sends data to the bus.
Receiver	The device which receives data from the bus.
Master	The device which initiates a transfer, generates clock signals, and terminates a transfer.
Slave	The device addressed by the master.
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message.
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted.
Synchronization	Procedure to synchronize the clock signals of two or more devices.

Source: The I^{2}C Bus Specification, Version 2.1, page 7.

The most interesting feature is the multimaster concept in which several I^{2}C devices may try to be a master at the same time. The I^{2}C protocol handles this through "synchronization" and "arbitration." Synchronization is the process by which masters all use the same clock. It relies on the fact that the clock line (SCL) is pulled high by a pull-up resistor. The result is that the SCL line value is the wired-AND of all the SCL connections from the various I^{2}C devices on the bus. Masters generate their own clock during data transfers. If two or more masters attempt a data transfer at the same time, they will all attempt to put their own clock on the SCL line. Each "would-be" master generating a clock is going to try to pull the clock line low for a period of time (the low period) and then let the bus be pulled high for a period of time (the high period). They each have internal timers metering out these periods of time. Different devices may have slightly different times for the low periods and high periods. The first clock to go from high to low "resets" the clock-generating circuitry of all the other "would-be" masters and starts them all counting out their low period. The device with the longest period will still be holding the SCL line low when all the other devices have released it to go high. That device will determine the low period of the SCL line.

When the last device releases the SCL line and it goes from low to high, all the devices now start counting out their high period. The first device to pull the SCL line from high to low causes the entire process to be repeated. The resulting waveform on the SCL line is what is called the synchronized clock. It has a low period equal to the longest low period of all the "would-be" masters and a high period equal to the shortest high period of all the "would-be" masters.

The process of arbitration is very similar. The synchronization process has not determined which device is the master—it has only determined which device has defined a clock they can all agree upon. The SDA (serial data) line is also a wired-AND, this time of all the individual SDA values. As each would-be master is participating in generating the clock, it is also putting data (in the form of individual bits) on the SDA line. If one device puts out a logic 1 while another device outputs a logic 0, the logic 1 is eliminated by the wired AND. Any device whose bit is eliminated by a wired AND loses the arbitration and will not become the master during this data transfer. So the last device to put out a logic 1 on the SDA line wins the arbitration and takes command of the bus, which means it completes the data transfer that's already been started.

The I^{2}C data format

There are a number of different elements that make up the bit format on the I^{2}C bus. These are:

- The start condition
- The address

- The read/write bit
- The acknowledge or not acknowledge bit
- The data
- The stop condition

The start condition

The start condition indicates the beginning of communication by a master. It occurs when the master pulls the data line (SDA) from high to low while the clock is high. The start condition always comes from the master.

The address

The address consists of a series of bits (7 bits in the examples we'll discuss) and each bit must be valid before the rising edge of the clock and be held valid until after the falling edge of the clock. The address bits always come from the master.

The read/write bit

The read/write bit occurs immediately after the address bits. Like the address bits, it must be valid before the clock goes high and held valid until the clock goes low. The read/write bit always comes from the master.

Table 11-2: Table showing meaning of the read/write bit

Mode	Read/Write Bit Value
Read	1
Write	0

The acknowledge bit

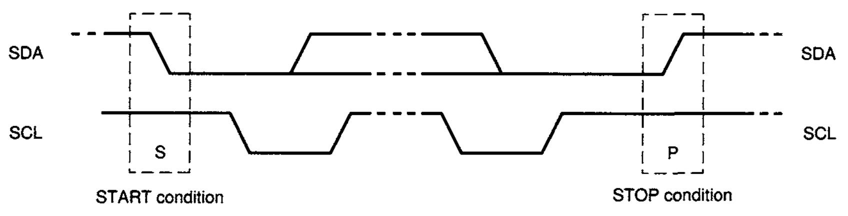
The acknowledge bit is used by both the master and slave to indicate continued responses to communication. Like the address and R/W bits, it must "overlap" the clock. Its exact use will be illustrated in more detail when we take a detailed look at the communications between master and slave. Both the master and slaves can produce an acknowledge bit.

Data bits

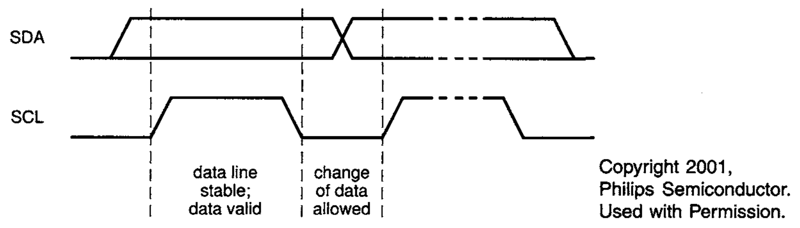
Data occurs in 8-bit chunks and after each chunk an acknowledge bit is issued. Data must overlap clock in the same fashion as the address, R/W and acknowledge bits.

The stop condition

The stop condition ends the communication and indicates that the bus is free. To generate a stop condition, the master lets the bus be pulled high while the clock is high. The stop condition is always generated by the master.



START and STOP conditions


SDA
SCL
data line stable; data valid
change of data allowed
Copyright 2001, Philips Semiconductor. Used with Permission.

Bit transfer on the I²C-bus

Figure 11-1: Diagram showing start, stop, and data with respect to clock

The data format for basic I²C communication using 7-bit addressing comes in three very similar configurations, corresponding to three possible actions:

- *Master writing to slave.* The process of the bus master writing to a receiving slave device proceeds as follows:

  a) The master issues a start condition.

  b) The master writes the 7-bit address to the bus.

  c) The master issues 1-bit R/W indicator (0 for write).

  d) The slave issues an acknowledge bit (logic 1).

  e) The master issues 8-bit data chunks, each followed by a 1-bit acknowledge from the slave.

  f) This continues until the master issues a stop condition.

- *Master reading from slave.* The process of the bus master reading from the slave proceeds much like that of writing to the slave.

  a) The master issues a start condition.

  b) The master writes the 7-bit address to the bus.

  c) The master issues a read/write bit (1 for read).

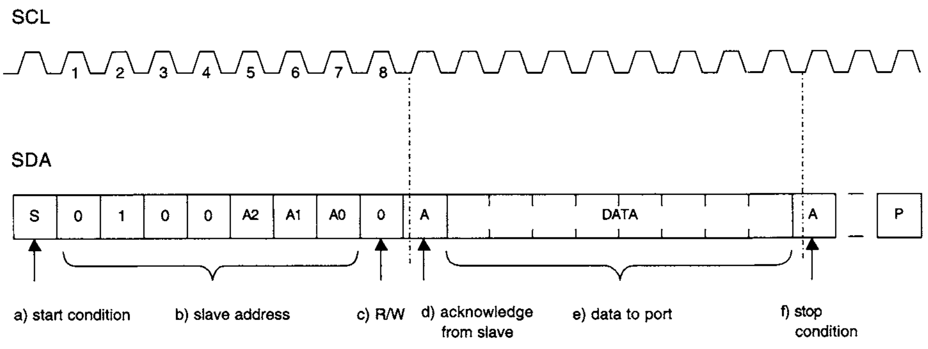


Figure 11-2: I²C Write

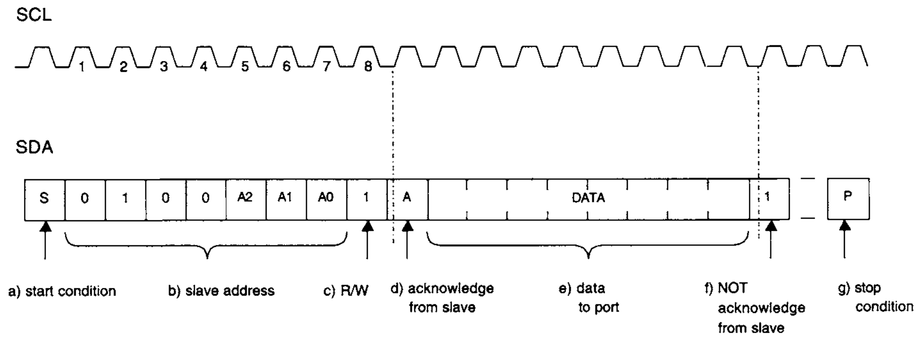


Figure 11-3: I²C Read

d) The slave issues an acknowledge bit (logic 1).

e) The slave outputs 8-bit data chunks, each of which is acknowledged by the master with an acknowledge bit (logic 1).

f) When the master is done reading, it will output a "not acknowledge" (logic 0).

g) Followed by the stop condition.

- *Master doing one, then doing the other.* When the master both reads from and writes to a slave, the result is what is referred to as the combined format. Like the name implies, it's basically a combination of the two previous examples. If we were to first write, then read, our process would be:

  a) Master issues a start condition.

  b) Master writes the 7-bit address, followed by a read/write bit (0 for write).

  c) The slave issues an acknowledge bit (logic 1).

d) Master writes 8-bit chunks, and after each chunk, the slave issues an acknowledge bit (logic 1).

e) When the master is done writing, and now wants to read, it reissues the start condition, reissues the address, and asserts the read bit (logic 1).

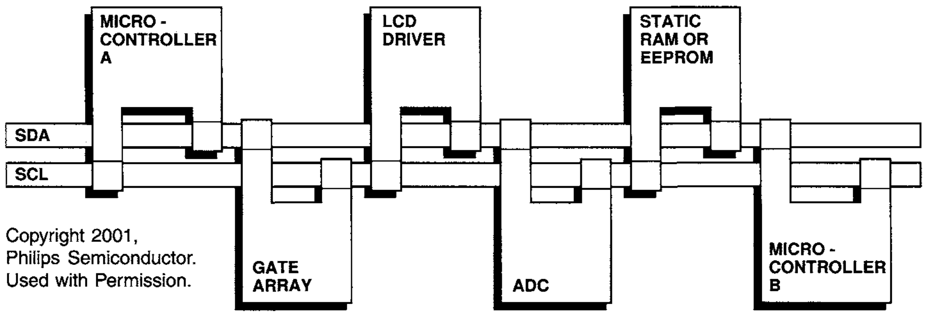
f) The slave will respond with an acknowledge bit (logic 1) followed by 8-bit data chunks. After each 8 bits of data, the master will issue an acknowledge bit (logic 1). When the master is done reading, it will issue a "not acknowledge" (logic 0) and a stop condition.

A few words about addressing

We have mentioned earlier that devices on the bus have a unique address. This has a somewhat different meaning than the "addresses" discussed in the section on the 1-Wire bus protocol. An I²C product such as the SAA1064[2], which is an I²C compatible seven-segment LED display driver, has a 7-bit address, 5 bits of which are the same for all SAA1064 devices and 2 of which are user configurable. Thus, there are 5 fixed and two programmable bits, respectively. Since two are programmable, you can have 2^2 or four SAA1064 devices on a single I²C bus. Other device types may have more or less fixed and programmable address bits (still adding up to seven). On the SAA1064, the 2 configurable bits are controlled by assigning a voltage to an external pin.

A typical I²C bus configuration

The most basic type of bus configuration for I²C is shown in Figure 11-4.



Example of an I²C-bus configuration using two microcontrollers

Figure 11-4: Drawing illustrating the basic I²C bus configuration

[2] SAA1064 Data Sheet – http://www.semiconductors.philips.com/pip/saa1064t

Extensions to the basic concept

The I²C protocol has many more features and extensions than described here, such as faster speeds, 10-bit addressing, and multivoltage pull-up configurations. We're not going to elaborate on them here. The best source for the complete story on I²C is the Philips I²C specification available on the Philips web site.

How TINI Does I²C

This next section examines how TINI communicates with devices over the I²C bus. We'll talk first about the hardware involved and then discuss the `I2Cport` class in the TINI API.

TINI and I²C: Hardware

There are a couple of ways in which TINI can generate the I²C data (SDA) and clock (SCL) signals. The primary way is through the direct use of two specific pins on one of the microcontroller data ports. But TINI is also capable of generating I²C signals through memory-mapped I/O, where one bit of a specific memory location is used to generate SDA, and one bit from another memory location is used to generate SCL.

Direct use of microcontroller port pins for I²C

The simplest way to access I²C with TINI is through the use of Port #5, bit0 and bit1. These are often referred to as P5.0 and P5.1. These pins don't always use the I²C protocol. In fact, they are actually designed to communicate using the CAN bus protocol. But, when used in conjunction with the `I2Cport` class found in the TINI API, they act as an I²C bus. These correspond to pins 21 and 20 on the TINI microcontroller and pins 10 (labeled CTX) and 11 (labeled CRX) on the TINI edge card connector. The CAN ports, such as Port 5 on the microcontroller, already have weak pull-ups built into them, so when using them to communicate I²C we haven't found the need to put pull-ups on the SDA and SCL lines. If you experiment with TINI and I²C and you find that the bus isn't being pulled high, your application may benefit from a stronger pull-up on the bus (20kΩ resistors to Vcc will do nicely).

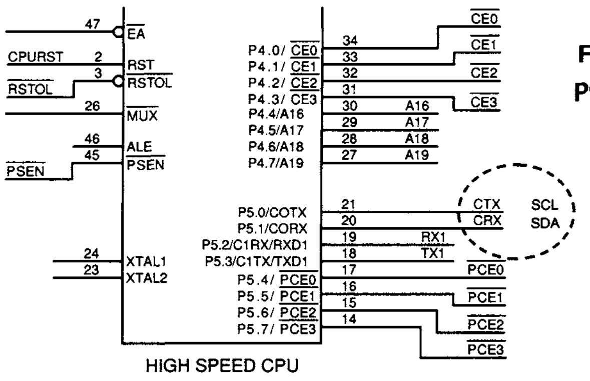


Figure 11-5: Drawing highlighting port 5 on the TINI microcontroller

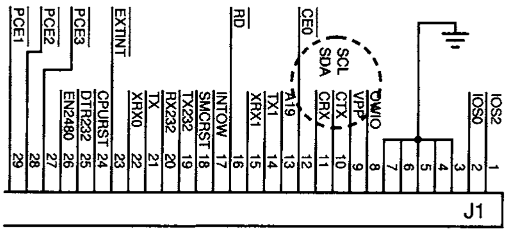


Figure 11-6:
Drawing of edge card connector

Memory-mapped driver for I²C

Another way to access I²C with TINI is through a memory-mapped driver. The circuit in the figure below shows the necessary hardware for this. The circuit uses a 74ACT138 address decoder to select the logic chips when the CPU is reading or writing I²C information. This is connected in the exact same manner as the address decoder we used in Chapter 8 for connecting other memory-mapped devices. The 74ACT244 input buffer is only enabled on data reads and the 74ACT753 latch is wired so it is enabled on data writes. These two chips are connected so that we have a bidirectional data flow between the data bus and the I²C bus and so that the input buffer reads the data on the outputs of the latch. This is necessary so we can make this a bi-directional bus and we can read data from I²C slave devices.

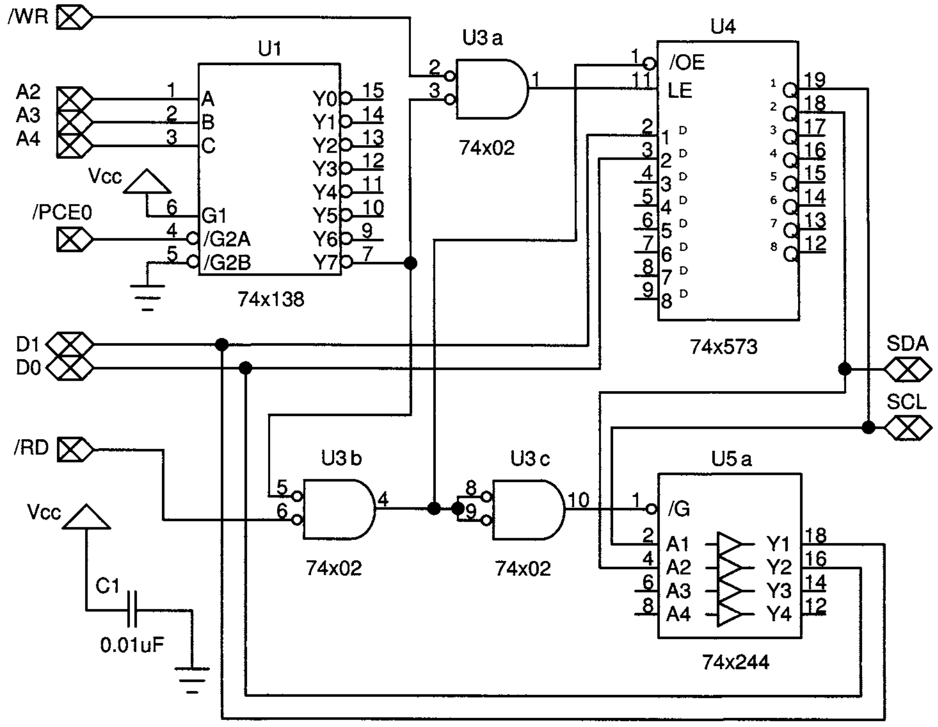


Figure 11-7: Memory-mapped I²C driver

TINI and I^2C: Software

The TINI API contains a java class, `I2CPort`, which can be used to generate the proper I^2C signals. The `I2CPort` class has an overloaded constructor which provides for two ways of constructing `I2CPort` objects. These correspond to the two ways of generating I^2C signals with TINI: directly using P5.0 and P5.1 or through memory-mapped I/O. For direct use of P5.0 and P5.1 the I2Cport constructor takes no arguments. For memory-mapped I^2C drivers, the constructor requires that you pass the address of the address decoder and a mask that tells the API which data line is to drive the SCL line and which data line is to drive the SDA line. Below is a list of the methods that we will use to talk I^2C on TINI.

- `public void` **`setAddress`**`(byte address)`
- `public void` **`setClockDelay`**`(byte delay)`
- `public int` **`write`**`(byte[] dataArray, int offset, int length)`
- `public int` **`read`**`(byte[] dataArray, int offset, int length)`

`setAddress()`

The `setAddress()` method accepts a byte representing the address of the I^2C device we are communicating with. It's important to note that the address we provide is the 7-bit device address *right justified*. So, the 7 bits of our address make up the 7 least significant digits of this byte, and the most significant bit is a 0. This is somewhat different than the address byte that actually gets sent out on the I^2C bus. That byte will be *left justified* (dropping the most significant bit which is 0) and the least significant bit will be the read/write bit. The conversion between the 7-bit address we provide the `setAddress()` method and the actual address byte sent on the I^2C bus is done for us by the `I2CPort` class.

`setClockDelay()`

The `setClockDelay()` method is used to modify the frequency of the SCL line. It accepts a byte as an argument, representing an additional increment of delay between all edges. When using the P5.0 and P5.1 pins for access to I^2C, our frequency is said to vary between 2.5 kbits/sec and 250 kbits/second. The documentation accompanying the API notes that each increment of clock delay byte adds .109 µsec between the clock edges. So a larger value as an argument to the `setClockDelay()` method leads to a slower clock, and vice versa. Dallas Semiconductor, in their TINI news group, has provided the following relationship between the clock period and the clock delay argument:

$$\text{SCL period} = (1\mu s + .109\mu s * \text{clock delay})*4$$

Experimentally, we've noticed that the relationship is more like this:

$$\text{SCL period} = \text{clock delay}*1.25\ \mu s + 5\ \mu s$$

This was derived from testing clock delay values of 2, 4, 8, 12, 14, 24, and 127. It should also be noted that the value in our test circuit (presented at the end of this section) worked for all of those values. The discrepancy between the clock period that we see and what TINI should be producing according to the documentation may have to do with the specifics of our setup.

`write()`

The `write()` method writes an array of bytes to the slave device currently being addressed. It takes the byte array as an argument, along with an offset and a length. The offset indicates where in the array you wish to begin the write and the length indicates how many bytes after the offset you wish to write. It's important to note that when constructing the array, you should not put the address byte into the array. The addressing is handled by the `setAddress()` method. The method returns a 0 if it received an acknowledge from the slave and a –1 if it didn't.

`read()`

The `read()` method behaves very much like the `write()` method. It reads data from the currently addressed slave and places it into an array. It takes the byte array as an argument, along with an offset and a length. The offset indicates where in the array you wish to begin placing the data and the length indicates how many bytes after the offset you wish to put there. The TINI 1.02 API notes that the method returns the number of bytes read on success, or –1 if it fails to receive an acknowledge from the slave. This appears to be a typo in the Javadocs. It actually returns 0 on success, like the `write()` method.

The process of connecting TINI to an I²C device and communicating is best illustrated with an example. For our example, we're going to connect an I²C-compatible 7-segment LED driver to TINI. We'll make a generic Java class that allows us to display digits, then use that class with one of our earlier example classes, `Thermometer`, to create a standalone TINI digital thermometer. This and all of the examples in this chapter will use the microprocessor ports pins for I²C communication. If you wish to use memory-mapped I²C then you simply need to use an alternate form of the constructor. Here are the two different forms of constructors:

- For Microcontroller Port driven I²C, use the following constructor:
  `LEDPort = new I2CPort();`
  This simply creates and returns a new I2CPort.

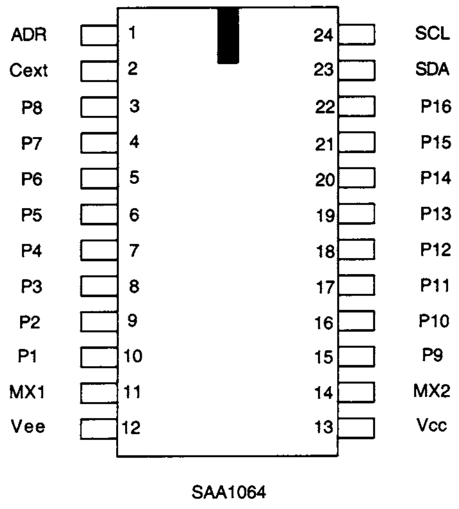
- For memory-mapped I²C following the schematic shown above, use the following constructor:
  `LEDPort = new I2CPort(0x0080001C, (byte)0x1, 0x0080001C, (byte)0x2 );`

This creates and returns a new I2CPort based on a memory-mapped driver. The address decoder is wired to decode the line driver and latch at 0x80001C in TINI memory and designates data lines D0 to drive SCL and D1 to drive SDA.

Example: Using TINI and I²C to drive a 7-segment LED display

We've chosen this as an example because it's easy to do, while illustrating all of the features key to understanding the TINI to I²C communication link. This example also shows an alternative way to implement an LED display from what we discussed in Chapter 8. We'll start with a quick discussion of the I²C-compatible 7-segment LED driver, the Philips SAA1064.

The SAA1064 is available in a 24-pin DIP package. It provides two 8-bit busses that can each handle one or two 7-segment LED displays. If they each are handling two displays, they are to be multiplexed. For simplicity, we're going to make a nonmultiplexed display in which each 8-bit bus drives a single 7-segment display. You can build the 4-digit display by following the schematic provided in the SAA1064 datasheet. The device pinout and descriptions of the pins are shown in Figure 11-8.



SAA1064 Data Sheet, dated feb 1991, page 4

Figure 11-8: SAA1064 pinout

ADR (pin 1) – This pin is used to configure the programmable address bits of each device. The two least significant bits of the 7-bit address of the SAA1064 are determined by the voltage applied to the ADR pin.

Table 11-3: Table of ADR values

ADR Pin Value	A1	A0
V_{ee} to $(3/16)V_{cc}$	0	0
$(5/16)V_{cc}$ to $(7/16)V_{cc}$	0	1
$(9/16)V_{cc}$ to $(11/16)V_{cc}$	1	0
$(13/16)V_{cc}$ to V_{cc}	1	1

Note: 7 Bit Slave Address = 0 1 1 1 0 A1 A0
Source: Excerpt from SAA1064 Data Sheet, dated Feb 1991, page 9

C_{EXT} (pin 2) – This pin can be used to control the frequency at which we multiplex between two pairs of 7-segment displays. We won't be using this feature and will simply ground this pin.

P8-P1 (pins 3-10) – These pins are the data bus for one of the two data ports on the device. They each drive one segment of the 7-segment displays. They are designed to *sink* current, so the 7-segment displays need to be *common anode*. P9 is the MSB, P1 is the LSB.

MX1 (pin 11), MX2(pin 14) – These output pins can be used to power the anode of the LED displays, or to control transistors used to power them. They are switching outputs that switch at the frequency of the multiplexing oscillator. We will not be using them and will leave them unconnected.

Vee (pin 12) – This is the chip ground.

Vcc (pin 13) – This is the chip power. We will set it to 5V, but it has a maximum of 18V.

P9-P16 (pins 15-22) – These pins are the data bus for the second of the two data ports on the device. They each will drive one segment of the 7-segment displays. They are designed to *sink* current, so the 7-segment displays need to be *common anode*. P16 is the MSB, P9 is the LSB.

SDA (pins 23) – This is the I²C data line.

SCL (pins 24) – This is the I²C clock line.

We will connect each of the two data buses on the SAA1064 to a single common anode 7-segment display. Then we will connect the SDA line to TINI pin 11 (CRX) and the SCL line to TINI pin 10 (CTX). Since there are weak pull-ups in the TINI microcontroller on those pins, we're not going to put additional pull-ups on these lines. The schematic for this is shown in Figure 11-9.

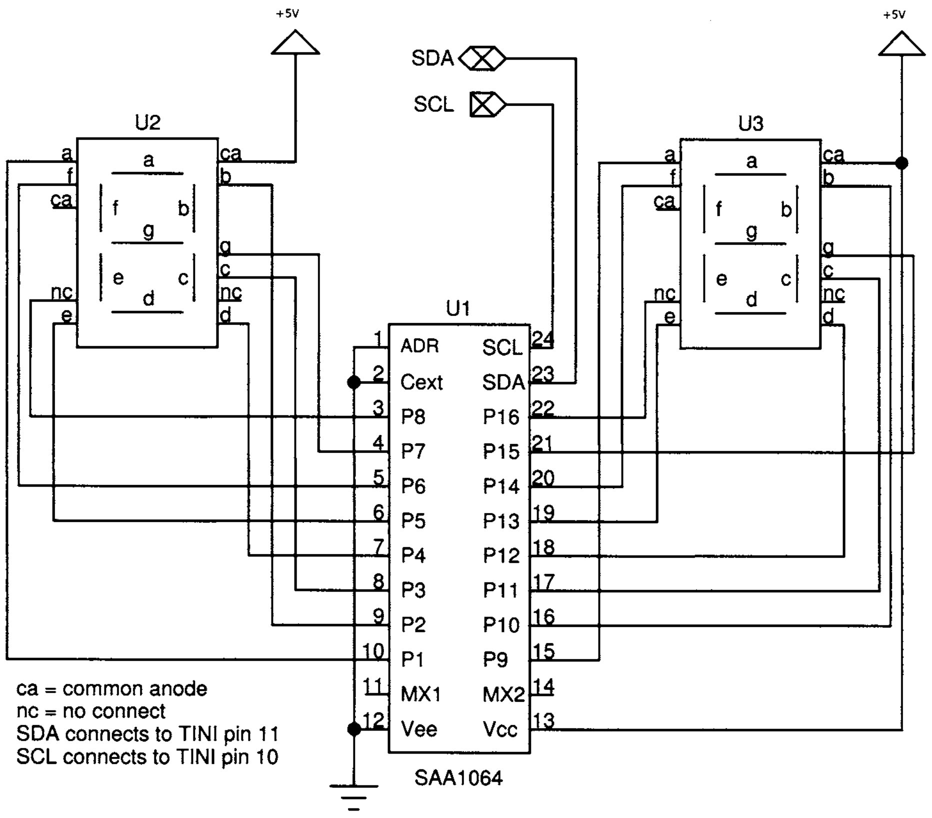


Figure 11-9: Schematic of the I²C LED

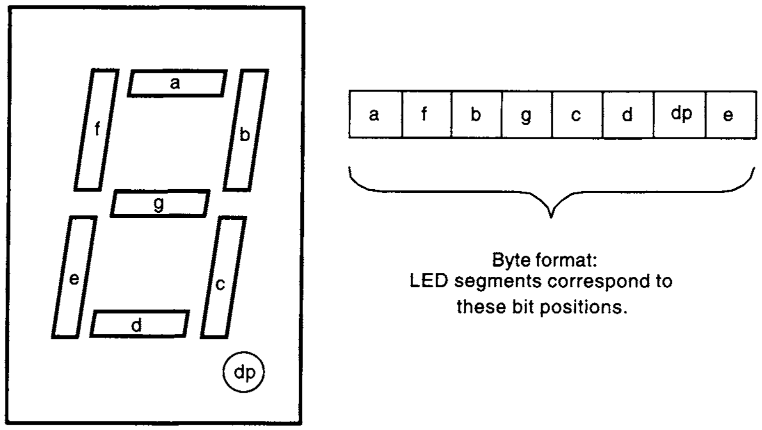


Figure 11-10: LED segments

For reference, a table and figure showing the mapping between bits on the data bus and the character that is displayed by those bits is also presented.

Table 11-4: Data bits to character displayed

Display Character	Output Bus (bits)	Output Bus (hex)
0	11101101	ED
1	00101000	28
2	10110101	B5
3	10111100	BC
4	01111000	78
5	11011100	DC
6	11011101	DD
7	10101000	A8
8	11111101	FD
9	11111100	FC
U	01101101	6D
F	11010001	D1

The next thing we need to consider is how to talk to the SAA1064.

SAA1064 data format

The process of displaying digits with the SAA1064 LED display driver is simple: you send it an array of data containing the commands and data. The array is seven bytes long, and the format is illustrated in the chart shown in Table 11-5.

Table 11-5: Chart showing SAA1064 data format

Data Byte	Byte Purpose	Byte Contents
1	Slave Address	0 1 1 1 0 A1 A0 Read/Write
2	Instruction Byte	0 0 0 0 0 SC SB SA
3	Control Byte	X C6 C5 C4 C3 C2 C1 C0
4	Digit 1 Data	D7 D6 D5 D6 D3 D2 D1 D0
5	Digit 2 Data	D7 D6 D5 D6 D3 D2 D1 D0
6	Digit 3 Data	D7 D6 D5 D6 D3 D2 D1 D0
7	Digit 4 Data	D7 D6 D5 D6 D3 D2 D1 D0

A1, A0 = configurable address bits.
SC, SB, SA = subaddress bits that are set to 0 for our purposes.
X = don't care (set to 0).
C6 – C0 are further explained in the text.
Source: Excerpt from the SAA1064 Data Sheet, dated Feb 1991, page 5.

The slave address

The slave address is seven bits long, with the most significant five bits fixed at 01110. The remaining two bits are programmable, based on the value of the ADR pin. We will tie our ADR pin low, which means the least significant two bits of our 7-bit address will be 00. Our 7-bit address makes up the "upper" seven bits of the first 8-bit byte in the data array that we send to the SAA1064. The eighth bit, the least significant bit of that byte, is the read/write bit. Since our data array will be written to the SAA1064, the read/write bit will be set to 0. Our slave address is 0111000, and the first data byte in our data array is 01110000, or 0x70.

The instruction byte

The instruction byte contains three subaddress bits, SC, SB, SA, which can be used to control where in the SAA1064 our data array gets written. We're not going to be making use of this feature, and all of our subaddressing bits are going to be set to zero. The second byte in the data array that we're going to send to the SAA1064 becomes 00000000.

The control byte

The control byte consists of an unused bit, which is the most significant bit of this byte, and seven control bits. The meaning of the seven control bits is shown below. Our control byte will be 0x26, or 00100110, which, from the table below, means that we are in static mode as opposed to multiplexed, sinking a current of 3 mA in each segment of the display.

Table 11-6: Chart showing SAA1064 control bytes

Bit Within Control Byte	Bit Meaning
C0	0 = static mode, constant display of digits 1 & 2
	1 = dynamic mode, digits 1 & 3 alternating with 2 & 4
C1	0 = digits 1 & 3 blanked
	1 = digits 1 & 3 not blanked
C2	0 = digits 2 & 4 blanked
	1 = digits 2 & 4 not blanked
C3	0 = normal operation
	1 = test mode, all segments lit
C4	0 = add no current
	1 = add 3 mA to the segment drive
C5	0 = add no current
	1 = add 6 mA to the segment drive
C6	0 = add no current
	1 = add 12 mA to the segment drive

Source: Excerpt from SAA1064 Datasheet, dated Feb. 1991, page 6.

Data bytes

The four data bytes each represent one of the four possible 7-segment LED displays that we can control with the SAA1064. Since we aren't using it in the dynamic, or multiplexed, mode, we are only concerned with two of the digits. In the static, two-digit mode, we only need to be concerned with the contents of the first two bytes. The last two bytes we set to 0x00. The coding that relates how the individual bits map into illuminated segments is specific to the individual design. Our encoding is illustrated in Table 11-4.

With that as background, if we wanted to display the number "38" on our system, we would write the following data array to the device: 0x70, 0x00, 0x26, 0xBC, 0xFD, 0x00, 0x00. Let's take a look at a generic Java class called `Digits` that takes in an integer and displays it on an SAA1064 controlled by TINI. We'll present the whole program, then study it in detail. It should be noted that making I²C work with TINI is actually quite simple, but it requires attention to detail. The `I2Cport` class handles some of the low-level details for us, which is a good thing. But it also means that the device address and the data array that we send the device have two different contexts. There's the address and data array that we use in our Java program, and then there's the address byte and the data array that gets transmitted on the I2C bus itself. They're not the same.

Our `Digits` class is designed to take in a two-digit number and display it on two 7-segment LED displays. If the number is less than 0, it displays "UF" for "underflow" and if the number is more than 99, it displays "OF" for "overflow." The class has a method to display a byte, and a method to turn the digits off. It has a `main()` method that acts as a test, displaying all of the digits.

Listing 11-1: Digits.java

```
import com.dalsemi.system.*;
import java.io.*;

public class Digits {

     private static byte LETTER_U = (byte) 0x6D;
     private static byte LETTER_F = (byte) 0xD1;
     private static byte LETTER_O = (byte) 0xED;
     private static byte BLANK = 0x00;

    byte[] data = new byte[6];
    I2CPort LEDPort;

    public Digits() {
       LEDPort = new I2CPort();
       LEDPort.setAddress((byte)0x38);
```

```
        LEDPort.setClockDelay((byte)0x7F);
        data[0] = 0x00;     // instruction byte
        data[1] = 0x26;     // control byte
        data[2] = 0x00;     // digit 1 data
        data[3] = 0x00;     // digit 2 data
        data[4] = 0x00;     // digit 3 data
        data[5] = 0x00;     // digit 4 data
    }


    public byte getBits(byte displayChar) {
        switch (displayChar) {
            case 1: return (byte)(0x28);
            case 2: return (byte)(0xB5);
            case 3: return (byte)(0xBC);
            case 4: return (byte)(0x78);
            case 5: return (byte)(0xDC);
            case 6: return (byte)(0xDD);
            case 7: return (byte)(0xA8);
            case 8: return (byte)(0xFD);
            case 9: return (byte)(0xFC);
            case 0: return (byte)(0xED);
            default: return (byte)(0);
        }
    }

    public void setValue(byte displayValue) {
        byte char1, char2;
        int i;
        if (displayValue < 0) {
              // Underflow (UF)
            char1 = LETTER_U;
            char2 = LETTER_F;
        }
         else if (displayValue > 99) {
              // Overflow (OF)
            char1 = LETTER_O;
            char2 = LETTER_F;
        }
         else {
            char2 = getBits((byte)(displayValue%10));
            char1 = getBits((byte)(((displayValue-(displayValue%10))/10)));
        }
        this.data[2] = char1;
        this.data[3] = char2;
        try {
            i = this.LEDPort.write(data, 0, 6);
              System.out.print( "Stat: " + i );
        } catch (Exception e) {System.out.println(e);}
    }
    public void turnOff() {
        int i;
        this.data[2] = BLANK;
        this.data[3] = BLANK;
        try {
            i = this.LEDPort.write(data, 0, 6);
        } catch (Exception e) {System.out.println(e);}
    }

    public static void main(String[] args) {
        Digits displayChars = new Digits();

        for (int i=-5; i<105; i++) {
            System.out.print( "N: " + i + " " );
            displayChars.setValue((byte)i);
            try {
                Thread.sleep(500);
            } catch(Exception e){}
        }
        displayChars.turnOff();
    }
}
```

The program begins with a couple of import statements, gaining access to the necessary Java class libraries.

```
import com.dalsemi.system.*;
import java.io.*;
```

Shown below is our class and data member declarations. The byte array data is only six bytes long. In our discussion of the SAA1064 we noted that the data array that gets sent to the device is seven bytes long. The difference comes from the fact that, in the I2Cport class, we don't have to explicitly put the address byte in the data array. Instead, we use the `setAddress()` method to set the address, and the class handles the details of putting the address into the data array. `LEDPort` is a `I2CPort` object that will give us access to the I2CPort methods.

```
public class Digits {

     private static byte LETTER_U = (byte) 0x6D;
     private static byte LETTER_F = (byte) 0xD1;
     private static byte LETTER_O = (byte) 0xED;
     private static byte BLANK = 0x00;

    byte[] data = new byte[6];
    I2CPort LEDPort;
```

Following is our constructor. It creates an `I2CPort` object, sets its device address, the clock delay, and initializes its data array. Here, we run into one source of confusion: the address. The actual device address is seven bits, and in our case it will be

0111000. The data field called `slaveAddress` in the `I2CPort` class, which is set by the `setAddress()` method, is a byte. Our 7-bit address is *right justified.* That is, they give it a leading zero, turning 0111000 into 00111000, or 0x38. When that address actually gets written out on the bus, it won't appear as 0x38. The I2CPort class extracts the actual 7-bit address from the byte, *left justifies* it, and makes the least significant bit the read/write bit, which in our case (writing), will be a 0. So, on the I^2C bus itself, the address byte will turn out to be 01110000, or 0x70. With respect to the clock delay, the value of 0x7F represents the slowest possible clock. Theoretically, the clock can vary between 2.5 kbits/sec and 250 kbits/sec. When measured with a scope, this example had a clock speed of 6 kbits/sec. In practice, with this example, this value hasn't proven to be critical. Experimentally, any value larger than 1 worked.

```
public Digits() {
    LEDPort = new I2CPort();
    LEDPort.setAddress((byte)0x38);
    LEDPort.setClockDelay((byte)0x7F);
    data[0] = 0x00;     // instruction byte
    data[1] = 0x26;     // control byte
    data[2] = 0x00;     // digit 1 data
    data[3] = 0x00;     // digit 2 data
    data[4] = 0x00;     // digit 3 data
    data[5] = 0x00;     // digit 4 data
}
```

The `getBits()` method takes a single digit between 0 and 9 and returns the 7-segment encoding for that digit. A bit value of "1" means that segment "a" will be illuminated. Refer to Table 11-4.

```
public byte getBits(byte displayChar) {
    switch (displayChar) {
        case 1: return (byte)(0x28);
        case 2: return (byte)(0xB5);
        case 3: return (byte)(0xBC);
        case 4: return (byte)(0x78);
        case 5: return (byte)(0xDC);
        case 6: return (byte)(0xDD);
        case 7: return (byte)(0xA8);
        case 8: return (byte)(0xFD);
        case 9: return (byte)(0xFC);
        case 0: return (byte)(0xED);
        default: return (byte)(0);
    }
}
```

The `setValue()` method takes in a byte that represents the 2-digit number we want to display, extracts the two digits from it, gets the bit encoding from the `getBits()`

method, and writes the data to the SAA1064 using the `write()` method of the `I2CPort` class. If the number is less than zero, it puts "UF" into the variables that will be displayed. If it's greater than 99, it puts "0F" into them. If between the two, it converts a number into two digits.

```
public void setValue(byte displayValue) {
    byte char1, char2;
    int i;
    if (displayValue < 0) {
         // Underflow (UF)
       char1 = LETTER_U;
       char2 = LETTER_F;
    }
     else if (displayValue > 99) {
         // Overflow (OF)
       char1 = LETTER_O;
       char2 = LETTER_F;
    }
     else {
       char2 = getBits((byte)(displayValue%10));
       char1 = getBits((byte)(((displayValue-(displayValue%10))/10)));
    }
```

We've established what characters to display, so we place them into the third and fourth byte positions in the data array. The data array is passed to the `write()` method with an offset of 0 and a length of six. The 0 offset indicates start with the 0 element in the array. The method returns an integer: 0 if the write functioned properly, and –1 if there was no Acknowledge bit sent from the slave.

```
    this.data[2] = char1;
    this.data[3] = char2;
    try {
       i = this.LEDPort.write(data, 0, 6);
         System.out.print( "Stat: " + i );
    } catch (Exception e) {System.out.println(e);}
}
```

The `turnOff()` method turns off all segments in both displays. We do this by setting all bits to 0 in both data bytes. Note that we're not telling them to write the digit 0, we're telling the SAA1064 to put 0s on all 8 bits of both data buses.

```
public void turnOff() {
    int i;
    this.data[2] = BLANK;
    this.data[3] = BLANK;
    try {
       i = this.LEDPort.write(data, 0, 6);
    } catch (Exception e) {System.out.println(e);}
}
```

The `main()` method tests the functionality of the `Digit` class. It simply counts up from –5 to 100. In doing so, it exercises all segments in both digits and tests for the correct encoding of all the numbers plus the "UF" (underflow) and "OF" (overflow) cases.

```
    public static void main(String[] args) {
        Digits displayChars = new Digits();
        for (int i=-5; i<101; i++) {
            displayChars.setValue((byte)i);
            try {
                Thread.sleep(500);
            } catch(Exception e){}
        }
        displayChars.turnOff();
    }
}
```

If you compile this into `Digits.tini`, then you will need to send it to TINI using FTP and execute it under the Slush operating system. If you compile it into `Digits.tbin`, then you would load it into TINI with JavaKit. This program provides no output to the screen, unless there is an exception. The commands used to compile this program into a `.tini` file are shown below.

```
C:\> javac -bootclasspath %TINI_HOME%\bin\tiniclasses.jar
    -classpath %TINI_HOME%\bin\owapi_dependencies_TINI.jar;.
    -d bin  src\Digits.java
C:\> java -classpath %TINI_HOME%\bin\tini.jar;. BuildDependency
    -p %TINI_HOME%\bin\owapi_dependencies_TINI_001.jar
    -f bin
    -x %TINI_HOME%\bin\owapi_dep.txt
    -o bin
    -d %TINI_HOME%\bin\tini.db
```

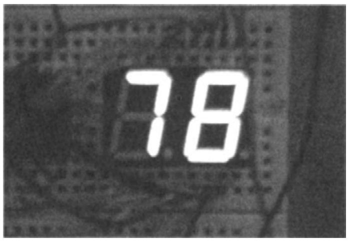


Figure 11-11: LED display in action

Chapter 7 as well as the thermometer example at the end of Chapter 10 has a discussion of what some of these options mean. Command options have been shown on separate lines for readability. In practice, they need to be on the same line.

Example: A TINI digital thermometer

The Digits class can be used in conjunction with the Thermometer class from an earlier example to make a simple thermometer that uses TINI, a DS1920 temperature iButton or a DS1820 1-Wire thermometer, and the SAA1064 display driver. Simply implement the previous example, and connect a thermometer iButton or 1-Wire device to TINI via the 1-Wire bus. We'll call the new class `LEDTherm`.

Listing 11-2: LEDTherm.java

```
import java.io.*;
public  class LEDTherm {
   public static void main (String[] args) {
      Thermometer therm = new Thermometer();
      Digits LEDs = new Digits();
      while (true) {
      try {
         therm.measureT();
         LEDs.setValue((byte)(therm.degF));
         Thread.sleep(1000);
         } catch(Exception e) {System.out.println(e);}
      }
   }
}
```

```
C:\> cd src
C:\> javac -bootclasspath %TINI_HOME%\bin\tiniclasses.jar
   -classpath %TINI_HOME%\bin\owapi_dependencies_TINI.jar;.
   -d ..\bin  Digits.java
C:\> cd ..
C:\> java -classpath %TINI_HOME%\bin\tini.jar;. BuildDependency
   -p %TINI_HOME%\bin\owapi_dependencies_TINI_001.jar
   -f bin
   -x %TINI_HOME%\bin\owapi_dep.txt
   -o bin
   -d %TINI_HOME%\bin\tini.db
```

When you run the program on your TINI you can watch the LED digits display the temperature.

Example: Extending TINI's parallel I/O

In the previous two examples we were writing to an I²C device. We could read back a status byte from the SA1064 LED driver, but this is not particularly interesting. This status byte contains a single 1-bit field that indicates that there was a power failure since the last time you read the status byte. A more interesting example is using a Philips PCF8574[3], remote 8-bit I/O expander, for adding 8-bit parallel I/O to your TINI.

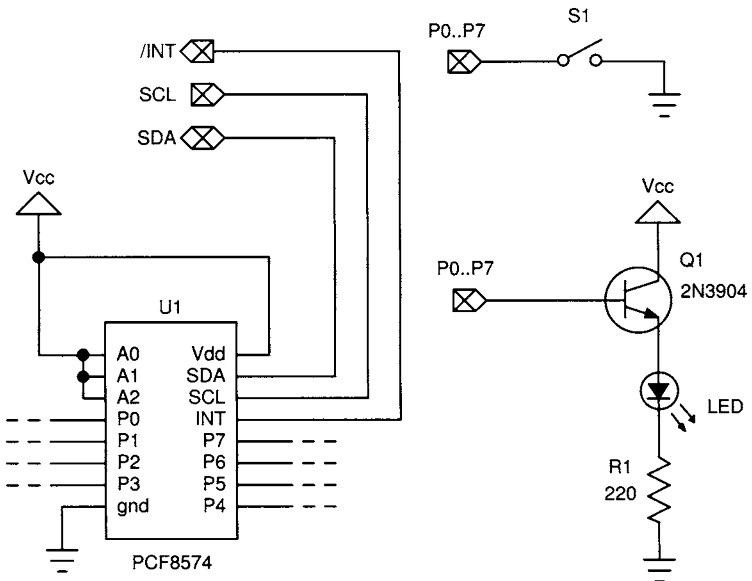


Figure 11-12: I²C 8-bit parallel I/O schematic

Listing 11-3: Parallel_IO.java

You can connect the switches and LEDs to any of the pins that you like. For the example Java program listed here, switches are connected to P0–P3 and LEDs are connected to P4–P7.

```
import com.dalsemi.system.*;
import java.io.*;

public class Parallel_IO {

   byte[] data = new byte[1];
   I2CPort PioPort;

   public Parallel_IO() {
      PioPort = new I2CPort();
      PioPort.setAddress((byte)0x27);
      PioPort.setClockDelay((byte)0x7F);
       data[0] = 0x00;
   }

    private static char[] hexChars =
       { '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F' };
```

[3] PCF8574 datasheet – http://www.semiconductors.philips.com/pip/pcf8574p

```
public static String toHex( byte data )
{
     StringBuffer output = new StringBuffer();
     int firstNibble, secondNibble;

          firstNibble = ( data >> 4 ) & 0x0F;
          secondNibble = data & 0x0F;
          output.append( hexChars[ firstNibble ] );
          output.append( hexChars[ secondNibble ] );

     return output.toString();
}

public void Blinky() {
     int stat=0;

     for (int i=1; i<10; i++) {
       System.out.print( "Write: " + i );

       try {
               data[0] = (byte)(0x1A);
               stat = PioPort.write( data, 0, 1);
               System.out.print( " [ " + toHex(data[0]) + " " + stat + " ]" );
            TINIOS.sleepProcess(250);

               data[0] = (byte)(0x2A);
               stat = PioPort.write( data, 0, 1);
               System.out.print( " [ " + toHex(data[0]) + " " + stat + " ]" );
            TINIOS.sleepProcess(250);

               data[0] = (byte)(0x4A);
               stat = PioPort.write( data, 0, 1);
               System.out.print( " [ " + toHex(data[0]) + " " + stat + " ]" );
            TINIOS.sleepProcess(250);

               System.out.println();
       }
          catch(Exception e){
               System.out.println( "Error in I2C write..." );
               System.out.println( e );
          }
   }


     for (int i=1; i<10; i++ ) {
          System.out.println( i );

          try {
               data[0] = (byte)(0x00);
```

```
                System.out.print( "read: " + i );
                stat = PioPort.read( data, 0,1 );
                System.out.println( " [ " + toHex(data[0]) + " " + stat + " ]" );
                TINIOS.sleepProcess(250);

            }
            catch(Exception e) {
                System.out.println( "Error in I2C read..." );
                System.out.println( e );
            }
        }

    }

    public static void main(String[] args) {
        Parallel_IO myPio = new Parallel_IO();
         myPio.Blinky();
    }
}
```

As with previous programs, the first few lines declare a data buffer for storing the data to be sent to, or received from, the I²C device, in this case a single byte. We have also created a single constructor that creates a new `I2CPort` object and assigns the address and clock delay.

```
import com.dalsemi.system.*;
import java.io.*;

public class Parallel_IO {

    byte[] data = new byte[1];
    I2CPort PioPort;

    public Parallel_IO() {
        PioPort = new I2CPort();
        PioPort.setAddress((byte)0x27);
        PioPort.setClockDelay((byte)0x7F);
         data[0] = 0x00;
    }
```

We also have a `toHex()` method for displaying the contents read from, or written to, the I²C device. This is entirely for our convenience, so we don't have to read decimal and determine if the output is proper. Then we have the bulk of the class in the `blinky()` method.

```
            try {
                data[0] = (byte)(0x1A);
                stat = PioPort.write( data, 0, 1);
                System.out.print( " [ " + toHex(data[0]) + " " + stat + " ]" );
                TINIOS.sleepProcess(250);
            ...

            System.out.println();
        }
        catch(Exception e){
            System.out.println( "Error in I2C write..." );
            System.out.println( e );
          }


      for (int i=1; i<10; i++ ) {
          System.out.println( i );

          try {
              data[0] = (byte)(0x00);
              System.out.print( "read: " + i );
              stat = PioPort.read( data, 0,1 );
              System.out.println( " [ " + toHex(data[0]) + " " + stat + " ]" );
              TINIOS.sleepProcess(250);

          }
          catch(Exception e) {
              System.out.println( "Error in I2C read..." );
              System.out.println( e );
          }
      }
```

The `blinky()` method does two things. First it writes various bytes to the PCF8574 to turn on and off some of the LEDs. Then it reads from the PCF8574 and displays the results of the various switch settings. We have liberally filled this program with lots of print statements so we can see what's happening along the way.

Compile the program and run it on TINI and watch the results.

```
C:\> javac -bootclasspath %TINI_HOME%\bin\tiniclasses.jar
    -classpath %TINI_HOME%\bin\owapi_dependencies_TINI.jar;.
    -d bin  src\Parallel_IO.java
C:\> java -classpath %TINI_HOME%\bin\tini.jar;. BuildDependency
    -p %TINI_HOME%\bin\owapi_dependencies_TINI_001.jar
    -f bin
    -x %TINI_HOME%\bin\owapi_dep.txt
    -o bin
    -d %TINI_HOME%\bin\tini.db
```

This next example is a slight extension of the last one. Instead of reading or writing from the same device, we will be reading inputs from one PC8574 and writing outputs to a second one. This will also demonstrate connecting several devices to an I²C bus at the same time. We will be using the first remote 8-bit I/O expander to read the position of eight push buttons and then writing this value to the second I/O expander to turn on or off eight LEDs. We will also use the interrupt feature of the first PCF8574 to trigger an external interrupt on a TINI stick when any of the buttons is pressed.

Listing 11-04: InOut.java

```
import java.util.TooManyListenersException;
import com.dalsemi.system.*;
import java.io.*;

class InOut implements ExternalInterruptEventListener {

    int i;
    byte[] data = new byte[1];

    I2CPort PioPort_I;
    I2CPort PioPort_O;

    private static char[] hexChars =
        { '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F' };

    public static String toHex( int data )
    {
        StringBuffer output = new StringBuffer();
        int firstNibble, secondNibble;

            firstNibble = ( data >> 4 ) & 0x0F;
            secondNibble = data & 0x0F;
            output.append( hexChars[ firstNibble ] );
            output.append( hexChars[ secondNibble ] );
        return output.toString();
    }

    public void init() throws TooManyListenersException
    {
        int stat=0;

        // This is the signal to which we will add an event listener
        ExternalInterrupt myInterrupt = new ExternalInterrupt();
        // Add the event listener
        myInterrupt.addEventListener(this);

        // Set this to EDGE triggering
        try {
            myInterrupt.setTrigger( true, this );
        }
        catch (Exception e){
            System.out.println( e );
        }
        // Set the addresses and clock delay
        PioPort_I = new I2CPort();
        PioPort_I.setAddress((byte)0x21);
        PioPort_I.setClockDelay((byte)0x7F);

        PioPort_O = new I2CPort();
        PioPort_O.setAddress((byte)0x22);
        PioPort_O.setClockDelay((byte)0x7F);

        // Set all outputs low
        try {
            data[0] = (byte)(0x00);
            stat = PioPort_I.write( data, 0,1 );
            stat = PioPort_O.write( data, 0, 1);
        }
        catch (Exception e) {
            System.out.println( e );
        }

    }

    public void externalInterruptEvent(ExternalInterruptEvent ev)
    {
        int stat = 0;

        System.out.println( "Interrupt Caught: " + ++i );

        try {
            // Fetch the Input states
            data[0] = (byte)(0x00);
            stat = PioPort_I.read( data, 0,1 );
            System.out.println( "Read [ " + toHex(data[0]) + " " + stat + " ]" );

            // Write the states to the Outputs on the other device
            stat = PioPort_O.write( data, 0, 1);
            System.out.println( "Write [ " + toHex(data[0]) + " " + stat + " ]" );

            // Reset inputs
            data[0] = (byte)0x00;
            stat = PioPort_I.write( data, 0,1 );
            System.out.println( "Reset [ " + toHex(data[0]) + " " + stat + " ]" );
        }
        catch (Exception e) {
            System.out.println( e );
        }

        // Die after 10 pushes
        if (i > 9) { System.exit(0); }
    }
    public static void main(String[] args) throws TooManyListenersException
    {
        // Start up the InterruptListender
        InOut interrupt = new InOut();
        // Initialize everything
        interrupt.init();

        // Hang out for awhile
        while (true) {
            // do nothing
        }
    }
}
```

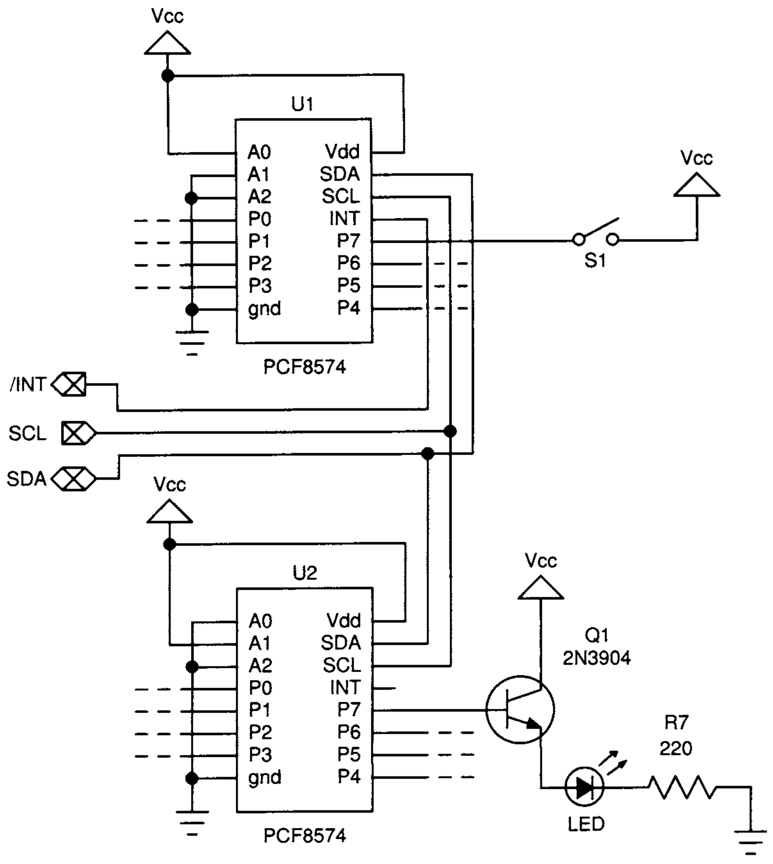


Figure 11-13: I²C 8-bit dual parallel I/O schematic

The program for this schematic is a slight modification of the previous, combined with what we learned in Chapter 8 for setting up an `ExternalInterruptEventListener`. We will spend much less time going through this program, as it's very much like the program from Chapter 8, ExtInt.java.

We have created the `init()` method to create the `ExternalInterrupt` object and set the EventListener. We also needed to set the interrupt triggering to "edge triggering" so we can catch which button was pressed when we trigger an interrupt. Edge triggering will trigger an interrupt every time the button changes state. This is in contrast to "level triggering," the other form of interrupt trigger, where the `ExternalInterruptEventListener` is called continually until the button is released. In this method we also create two `I2CPort` objects, one for each of the I²C devices, and we initialize them to all bits off.

```
public void init() throws TooManyListenersException
{
    int stat=0;

    // This is the signal to which we will add an event listener
    ExternalInterrupt myInterrupt = new ExternalInterrupt();
    // Add the event listener
    myInterrupt.addEventListener(this);

    // Set this to EDGE triggering
    try {
        myInterrupt.setTrigger( true, this );
    }
    catch (Exception e){
        System.out.println( e );
    }

    // Set the addresses and clock delay
    PioPort_I = new I2CPort();
    PioPort_I.setAddress((byte)0x21);
    PioPort_I.setClockDelay((byte)0x7F);

    PioPort_O = new I2CPort();
    PioPort_O.setAddress((byte)0x22);
    PioPort_O.setClockDelay((byte)0x7F);

    // Set all outputs low
    try {
        data[0] = (byte)(0x00);
        stat = PioPort_I.write( data, 0,1 );
        stat = PioPort_O.write( data, 0, 1);
    }
    catch (Exception e) {
        System.out.println( e );
        }
    }
```

The last thing before our main starts the whole thing running is the `externalInterruptEvent()` method. Each time the first PCF8574 fires an interrupt, this method is called. In this method we read the byte from the input device and write this to the output device to turn on or off the corresponding LEDs to match the buttons pushed. As with the previous example, this program is quite verbose so we can see what is going on in case the LEDs don't work as expected.

```
    public void externalInterruptEvent(ExternalInterruptEvent ev)
    {
        int stat = 0;

        System.out.println( "Interrupt Caught: " + ++i );

        try {
            // Fetch the Input states
            data[0] = (byte)(0x00);
            stat = PioPort_I.read( data, 0,1 );
            System.out.println( "Read [ " + toHex(data[0]) + " " +
                stat + " ]" );

            // Write the states to the Outputs on the other device
            stat = PioPort_O.write( data, 0, 1);
            System.out.println( "Write [ " + toHex(data[0]) + " " +
                stat + " ]" );

            // Reset inputs
            data[0] = (byte)0x00;
            stat = PioPort_I.write( data, 0,1 );
            System.out.println( "Reset [ " + toHex(data[0]) + " " +
                stat + " ]" );
        }
        catch (Exception e) {
            System.out.println( e );
        }

        // Die after 10 pushes
        if (i > 9) { System.exit(0); }
    }
```

To compile this:

```
C:\> javac -bootclasspath %TINI_HOME%\bin\tiniclasses.jar
    -classpath %TINI_HOME%\bin\owapi_dependencies_TINI.jar;.
    -d bin  src\InOut.java
C:\> java -classpath %TINI_HOME%\bin\tini.jar;. BuildDependency
    -p %TINI_HOME%\bin\owapi_dependencies_TINI_001.jar
    -f bin
    -x %TINI_HOME%\bin\owapi_dep.txt
    -o bin
    -d %TINI_HOME%\bin\tini.db
```

After you compile this and FTP it to your TINI, give it a test run. If you see lots of -1s printed in the output, this is probably because there is something not quite right with the way you have connected your I²C bus or devices, so check the schematics again. While it's hard to show the LEDs lighting, the screen output looks like this:

```
Interrupt Caught: 1
Read [ 01 0 ]
Write [ 01 0 ]
Reset [ 00 0 ]
Interrupt Caught: 2
Read [ 00 0 ]
Write [ 00 0 ]
Reset [ 00 0 ]
...
Interrupt Caught: 6
Read [ 81 0 ]
Write [ 81 0 ]
Reset [ 00 0 ]
Interrupt Caught: 7
Read [ 00 0 ]
Write [ 00 0 ]
Reset [ 00 0 ]
...
Interrupt Caught: 10
Read [ 80 0 ]
Write [ 80 0 ]
Reset [ 00 0 ]
```

Here you can see that we press the first button (01), then both the first and last button (81 is button 8 and button 1), then just the last button (80). Notice that each interrupt is followed with another that reads 00. This is the button release (it's changing state on the device and so it triggers another interrupt).

All of the examples in this chapter have used the microprocessor port driver for I²C communication. If you wish to use memory-mapped I²C then you simply need to use an alternate form of the constructor given in the discussion of the API. The rest of the example programs do not need any further modification.

Summary

The section has provided a very brief look at the I²C bus protocol and how to use it with TINI. There are a number of aspects of I²C and TINI that we have not attempted to cover here. The Dallas Semiconductor TINI archives are a rich source of information, in a question-and-answer format, for those interested in more information. Additionally, the Philips web site provides the complete I²C specification, free of charge.

References

1. *Philips Semiconductors, About the I^2C-bus,*
   http://www-us.semiconductors.philips.com/i2c/facts/
2. *Philips Semiconductors, I^2C Bus Specification,*
   http://www-us.semiconductors.philips.com/acrobat/various/
   I2C_BUS_SPECIFICATION_3.pdf
3. *I^2C FAQ,*
   http://perso.club-internet.fr/mbouget/i2c-faq.html
4. *The I^2C-bus and how to use it (including specification),*
   http://www.semiconductors.philips.com/acrobat/various/
   I2C_BUS_SPECIFICATION_1995.pdf