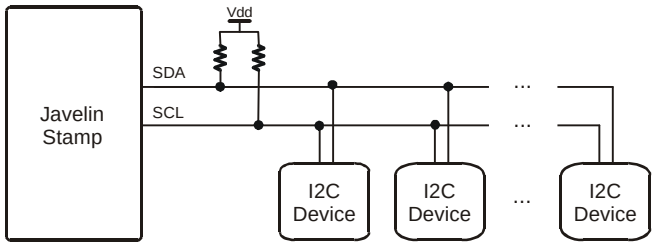
599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office/Tech Support: (916) 624-8333
Fax: (916) 624-8003

Web Site: www.javelinstamp.com
Home Page: www.parallaxinc.com

General: info@parallaxinc.com
Sales: sales@parallaxinc.com
Technical: javelintech@parallaxinc.com



Contents

Getting Started with I²C™ Devices – 24LC32 EEPROM Example

Phillips Semiconductor's I²C® devices probably have more to offer to the product designer than any other line of integrated circuits. These chips are designed to minimize the number of microcontroller I/O pins required by having many chips share the same bus (data and clock lines) as shown in the diagram at the top-right of this page. The list below shows some of the different functions I²C chips can perform, and for some products, this list could conceivably complete the design.

- Button/keypad readers
- LED/LCD controllers
- Time base and clock oscillators
- Phone tone transmitters and decoders
- Temperature sensors
- A/D and D/A converters
- I/O expanders
- EEPROM and RAM memory

With the help of the I2C library, you can program the Javelin Stamp to communicate with I²C and I²C compatible devices. This application note contains examples of how to translate datasheet documentation for an I²C device into Javelin Stamp code that makes use of the I2C library to communicate with the device. The specific chip used in these examples is the Microchip 24LC32, which is an I²C compatible device.

Downloads, Parts, and Equipment for the 24LC32

This application note along with its example program listings, library files, and javadoc html files are all available for free download from:

www.javelinstamp.com/Applications.htm

✓ Look for the section entitled "Getting Started with I2C" and download AppNote003–I2CPrimer.exe.

When you run this application, it will show you a window that looks similar to the one in Figure 3.1.

✓ If you installed the Javelin Stamp IDE to the default directory of C:\Program Files\Parallax Inc\Javelin Stamp IDE, click the Unzip button.

– OR –

If you installed the Javelin Stamp IDE to a different directory, use the Browse button to find the directory or hand-enter it into the Unzip to folder field before clicking Unzip.

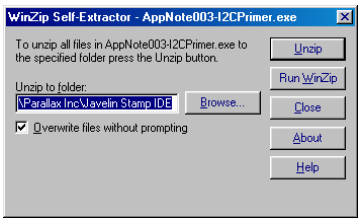


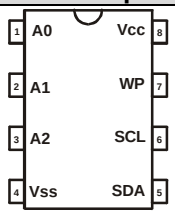
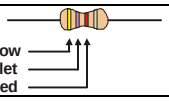
Figure 3.1
Unzipping the example files into your Javelin Stamp IDE.

Here is a list of the files that AppNote003-I2CPrimer.exe installs along with their locations relative to the Javelin Stamp IDE's folder. If you used the default install settings, these paths are inside C:\Program Files\Parallax Inc\Javelin Stamp IDE.

\doc\AppNote003_I2C_Primer_EEPROM_Example.pdf
\doc\I2C.pdf
\doc\MC24LC32LibEx.pdf
\lib\stamp\protocol\I2C.java
\lib\stamp\memory\eeprom\MC24LC32LibEx.java
\Projects\examples\protocol\i2cprimer\MC24LC32FindChips.java
\Projects\examples\protocol\i2cprimer\MC24LC32Demo.java
\Projects\examples\protocol\i2cprimer\MC24LC32LibExDemo.java
\Projects\examples\protocol\i2cprimer\TerminalHelper.java

Table 3.1 lists the parts you will need to take example Program Listings 3.1, 3.2 and 3.3 for a test drive. These example programs write data to and read data from a 24LC32. Two 4.7 kΩ pullup resistors are also required for this example. Regardless of how many I²C chips you place on a given bus, only two pullup resistors are required.

Table 3.1: Parts List

Quantity	Part Description	Pin Map
1	Parallax Part: 604-00020 Microchip 24LC32 4 k EEPROM	
2	Parallax Part: 150-04720 4.7 kΩ resistor	 

The equipment used to test this example included a Javelin Stamp, Javelin Stamp Demo Board, serial cable, 7.5 V, 1000 mA DC power supply, and PC with the Javelin Stamp IDE v2.01.

An I²C Bus Circuit and How it Works

Figure 3.2 shows a portion of an I²C bus with the example circuit we'll be working with. The portion shaded in gray is optional, and it's in the diagram to show how there could be more 24LC32 and other I²C chips connected to the bus. Above and beyond six more 24LC32s, you can connect other I²C chips, like a couple of PCF8574 I/O expanders for example.

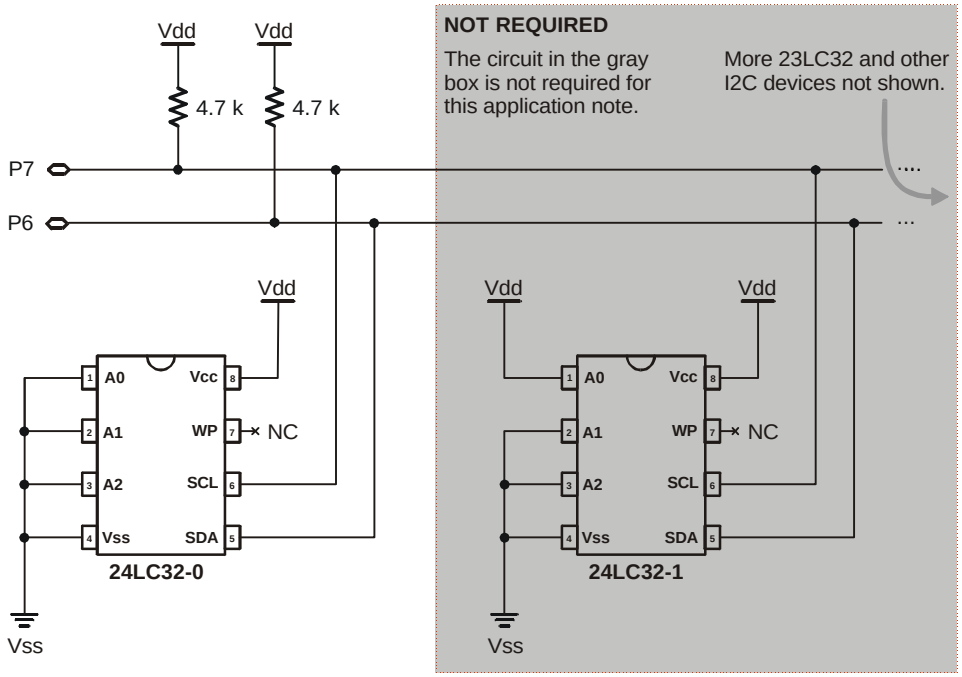


Figure 3.2 I²C Example Circuit

Each I²C chip on the bus listens for its unique slave address to be called out before it responds. This unique address is comprised of two parts, an external part that you can set by connecting certain pins to Vdd or Vss, and an internal part that is unique to the device. Different 24LC32s have the same internal address (binary 1010), which is different from a PCF8574's internal address of binary-0100. The reason you can have up to a total of eight 24LC32 chips on a given I²C bus is because there are eight different combinations of Vdd and Vss that you can connect to the chip's three external address pins (A2, A1, and A0). You can add up to eight more PCF8574 chips because the internal address differentiates it from any of the eight 24LC32 chips. Then, you can differentiate the PCF8574 chips from each other, again by wiring the A2, A1, and A0 pins differently for each PCF8574.

Single-Master Networks Only	Although the I²C protocol has provisions for networks with more than one microcontroller, the Javelin Stamp's I2C library is designed to work in a single-master network. This means that the Javelin Stamp is the only microcontroller connected to a given bus. The rest of the devices are called I2C slaves because they take orders from the I²C master (the Javelin Stamp).

The I2C Library and Data Exchanges with the I²C Bus

There are two steps to follow before you can access the I2C library file's fields and methods. First, import the I2C library:

```
import stamp.protocol.I2C;
```

Second, create an I2C object. Since the SDA line in this example connected to P6 and the SCL line is connected to P7, you can do it like this:

```
final public static int SDAPin = CPU.pin6;
final public static int SCLPin = CPU.pin7;
public static I2C i2cbus = new I2C(SDAPin, SCLPin);
```

Tip	Use the I2C javadoc (I2C.pdf in your \doc folder) as a reference to find out more about the **`I2C`**, library methods.

To initiate an information exchange, the Javelin Stamp sends a start condition. Using the I2C object we just declared, the start condition can be sent using the command:

```
i2cbus.start();
```

Next, the Javelin Stamp sends a first byte, which contains either part or all of the device's the slave address and a read/write value. The next section, entitled Calling Out the Address of an I²C Device, has a detailed example of how to put together this first byte. For the time being, let's say this byte turned out to be binary-10100001, which translates to hexadecimal-0xA1. After the first byte is sent by the Javelin Stamp, the slave is expected to reply with an acknowledge (ACK). With the I2C library, this all boils down to one method call that returns **`true`** if an acknowledge is received from the slave device or a **`false`** if no acknowledge (NAK) is received. If, earlier in the program, you declared a **`boolean`** field named **`reply`**, you can set **`reply`** equal to the I2C library's **`write`** method to see if the Javelin Stamp received an ACK or a NAK. The **`reply`** variable will be true if an ACK was received and false if a NAK was received.

```
reply = i2cbus.write(0xA1);
```

Certain I²C devices might require the Javelin Stamp to write a second "extended address byte". The device's data sheet will tell you if you have to send this second byte. If so, a second **`write`** operation that sends the value specified by the device's datasheet is required.

If, in the first byte, the Javelin Stamp set the read/write value to write, the Javelin Stamp will then write one or more values to the I²C device. The I²C slave device is expected to reply with an acknowledgement after each

byte it receives from the Javelin Stamp. For example, if an **int** value named **data** contains what you want to send in its lower byte, you can send it to the I²C device, using the **write** method:

```
reply = i2cbus.write(data);
```

Or, if you're not concerned with tracking the device's acknowledgement, just use:

```
i2cbus.write(data);
```

If, in the first byte, the Javelin Stamp set the read/write value to read, the Javelin Stamp reads one or more bytes from the I²C device. The Javelin Stamp replies with an acknowledge after each byte it receives from the slave. For example, if the data sheet requires the Javelin Stamp to send an ACK (**true**) after receiving the byte, you could use the command:

```
data = i2cbus.read(true);
```

If the data sheet requires that you send a NAK (**false**) after receiving the byte, you can use the command:

```
data = i2cbus.read(false);
```

In either case, after the command is executed, the variable **data** will contain the value sent by the I²C device.

FYI	The **I2C.read** method will also accept **I2C.ACK** or **I2C.NAK.**

The Javelin Stamp terminates an I²C transaction by sending a stop condition using the stop method:

```
i2cbus.stop();
```

Calling Out the Address of an I²C Device

Figure 3.3 shows the contents of the first byte that the Javelin Stamp must transmit to the I²C bus to initiate an exchange with a particular 24LC32. The Javelin Stamp has to send a start condition before sending the byte, and the slave device is expected to send an ACK if it got the message. Based on Figure 3.3, the byte that should be sent to 24LC32-0 for a read operation is 0xA1, and the byte that should be sent to the chip for a write operation is 0xA0. Here's why; the first byte contains three elements:

1) The upper four bits contain the I²C device's internal address bits, and it's referred to by the diagram as the control code. This value is binary-1010 or hexidecamal-0xA in the case of the 24LC32.
2) Bits 3, 2, and 1 contain the chip select values A2, A1, and A0. Because all of 24LC32-0's address pins are tied to Vss in the example circuit we are using, bits 3, 2, and 1 should all be set to zero.
3) Bit-0 is the read/write bit which is binary-1 for read and binary-0 for write.

When you combine these three elements into a single byte, it turns out to be binary-10100001 (hexadecimal – 0xA1) for a read or binary-10100000 (hexadecimal-0xA0) for a write operation.

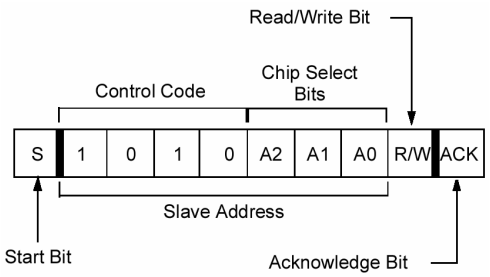


Figure 3.3
24LC32 data sheet excerpt that shows the contents of the byte sent to initiate an I²C data exchange.

The two commands below can be used to call out the address of the 24LC32-0 shown in Figure 3.2 and announcing that a write operation is about to happen.

```
i2cbus.start();
i2cbus.write(0xA0);
```

Program Listing 3.1 uses a more general approach which allows you to declare constants for the control code, chip select bits and read/write commands. By using the bitwise OR operator **|**, these terms can be combined together into a single byte before sending it using the I2C write method. This is approach makes larger programs and class files easier to manage. Keep in mind that you can use names of your choosing and add constants as needed. If you have different types of I²C devices on the same bus, each type of device will have a different control code. If you have more than one of the same type of device on a given bus, each will need a different slave address.

```
// From the declarations
final public static int CONTROL_CODE  = 0x00A0;
final public static int SLAVE_ADDRESS = 0x0000;
final public static int WRITE_BIT     = 0x0000;
final public static int READ_BIT      = 0X0001;

public static int controlByte;

// From the main code or a method
controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
i2cbus.start();
reply = i2cbus.write(controlByte);
```

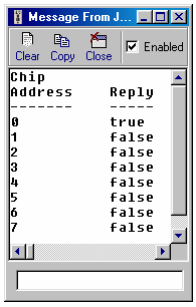
Because the **`bus.write`** method returns a **`boolean`** value, it can be used to see if an I²C chip is acknowledging at a particular address.
Program Listing 3.1 uses this fact to search for the different possible chip-select addresses that could exist on the I²C bus.

- ✓ Use the Javelin Stamp IDE to open MC24LC32FindChips.java in your examples\protocol\i2cprimer folder:
- ✓ Click the Run button in the Javelin Stamp IDE.

You can use
Program Listing 3.1 to test your circuit and make sure the 24LC32 is replying to the Javelin Stamp when it calls out the chip's I²C address. When you run MC24LC32FindChips.java, the output will resemble Figure 3.4. Note that only the 24LC32 with a chip address of 0 acknowledged.

- ✓ You can change the wiring to the chip select pins (A2, A1, and A0 in Figure 3.2), and the chip address with the ACK reply (**`true`**) will change when you re-run the program.
- ✓ If you experiment with the chip select wiring, make sure to rewire it to address-0 (A2, A1, and A0 tied to Vss) before moving on.

Figure 3.4
Messages from Javelin display from Program Listing 3.1.



Program Listing 3.1 – Search I²C Bus for 24LC32 Chips

```
package examples.protocol.i2cprimer;

import stamp.core.*;
import stamp.protocol.I2C;

public class MC24LC32FindChips {

  // Declare I2C object.
  final public static int SDAPin = CPU.pin6;
  final public static int SCLPin = CPU.pin7;
  public static I2C i2cbus = new I2C(SDAPin, SCLPin);
```

```
  // Declare constants for use with I2C class.
  final public static int READ_BIT      = 0x0001;
  final public static int WRITE_BIT     = 0x0000;
  final public static int CONTROL_CODE  = 0x00A0;
  final public static int SLAVE_ADDRESS = 0x0002;

  // Declare global variables.
  public static int controlByte;
  public static boolean reply;

  public static void main() {

    // Table heading.
    System.out.println("Chip ");
    System.out.println("Address    Reply");
    System.out.println("-------    -----");
    for(int slaveAddress = 0; slaveAddress < 8; slaveAddress ++){

      // I2C bus communication.
      controlByte = CONTROL_CODE | (slaveAddress << 1) | WRITE_BIT;
      i2cbus.start();
      reply = i2cbus.write(controlByte);

      // Display results.
      System.out.print(slaveAddress);
      System.out.print("          ");
      System.out.println( reply);
    }
  }
}
```

Four Examples of Writing Code for an I²C Device Using its Data Sheet

I²C device data sheets contain lots of information, and it's important to read a given I²C device's datasheet carefully before tinkering with the device. From the standpoint of making the device work with the Javelin Stamp, the three most important questions to answer are:

1) How do you wire up the device?
2) How does the device work?
3) How do you communicate with the device using:
    a. Start conditions
    b. Control bytes
    c. Read and write operations
    d. ACKs and NAKs
    e. Stop conditions

Question-3 can be rephrased as, how does the I²C master (the Javelin Stamp) exchange data with the slave device on the I²C bus (the 24LC32 in this example)? This section was written to help answer question-3, and it contains four data exchange examples that translate information from the 24LC32's data sheet into Javelin Stamp code. These examples cover the most common exchanges between a Javelin Stamp and an I²C slave device:

1 – Writing a Byte
2 – Reading a Byte
3 – Writing a string (of Bytes)
4 – Reading a string (of Bytes)

Example 1 - Writing a Byte

Figure 3.5 is another diagram from the 24LC32's data sheet that shows how to write a single byte to the EEPROM. For the 24LC32, it actually takes four write operations accomplish this. The start condition and the control byte should be familiar since that's what the previous program sent to search for 24LC32s. Figure 3.5 shows three more bytes, two that contain the EEPROM address and one that contains the data to be written to the EEPROM address. The entire data exchange is followed by a stop condition. All bytes should be acknowledged by the 24LC32. In terms of the I2C library, that means that the **`I2C.write`** method should return a **`boolean true`** after each write operation. After the four writes, the Javelin Stamp must send a stop condition.

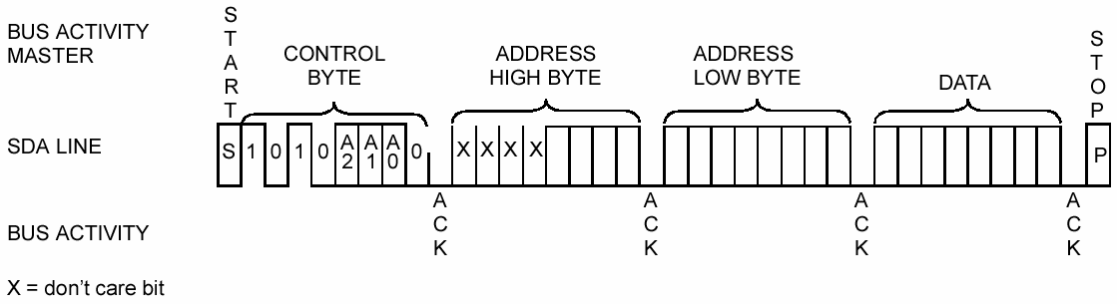


Figure 3.5 24LC32 datasheet excerpt on how to save a single byte to an address in the EEPROM.

Here is an example of how the Javelin Stamp can be programmed to execute the data exchange shown in Figure 3.5

```
// Set EEPROM address pointer (three writes).
controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
i2cbus.start();
```

```
i2cbus.write(controlByte);
i2cbus.write(eeAddress >> 8);
i2cbus.write(eeAddress);

// Write the byte (one more write).
i2cbus.write(data);
i2cbus.stop();
```

Example 2 - Reading a Byte

Figure 3.6 is another diagram excerpt from the 24LC32's data sheet that shows how to read a single byte from the EEPROM. The first three bytes set the EEPROM's address pointer just like in the previous example. Take a closer look at the two bytes to the right of the second start condition. They provide an example of a read operation. The only thing in the diagram that states that the "DATA BYTE" is the 24LC32's reply to a read operation is the fact that the "CONTROL BYTE" has a binary-1 in its rightmost or least significant bit (the read/write bit). Since this read/write bit is set to read (binary-1), the next byte is the I²C device's reply.

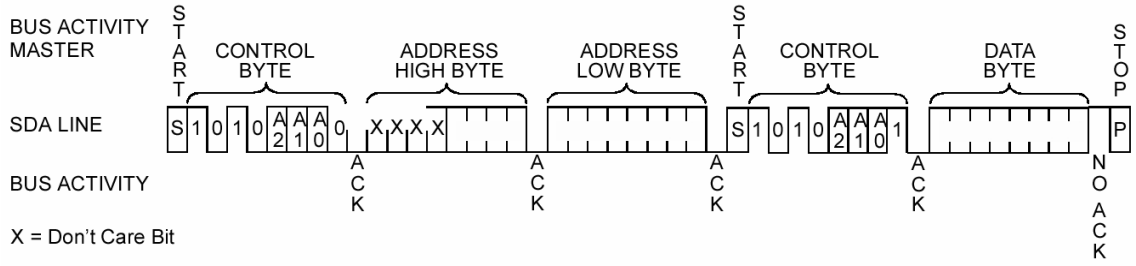


Figure 3.6 24LC32 datasheet excerpt on how to read a single byte from an address in the EEPROM.

In the previous example, the chip replied with either ACK or NAK in reply to a write operation. When performing a read operation, the Javelin Stamp has to send the ACK or NAK after the data is transmitted by the I²C chip. This is why the I2C library's read method expects a **`boolean`** parameter (either **`true`** for ACK or **`false`** for NAK). In this case, **`false`** is sent after receiving one byte of data using the **`I2C.read`** method. Here is an example of Javelin Stamp code that translates Figure 3.6 into an exchange between the Javelin Stamp and the 24LC32.

```
// Set EEPROM address pointer (three writes).
i2cbus.start();
controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
```

```
i2cbus.write(controlByte);
i2cbus.write(eeAddress >> 8);
i2cbus.write(eeAddress);

// Read operation sends a control byte (one write) and
// reads a byte (one read).
controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | READ_BIT;
i2cbus.start();
i2cbus.write(controlByte);
data = i2cbus.read(false);
i2cbus.stop();
```

Example 3 - Writing a String

I²C devices often have a built in feature that allows the master device to perform repetitive reads or writes. Figure 3.7 shows how the 24LC32's data sheet instructs you to do multiple write operations.

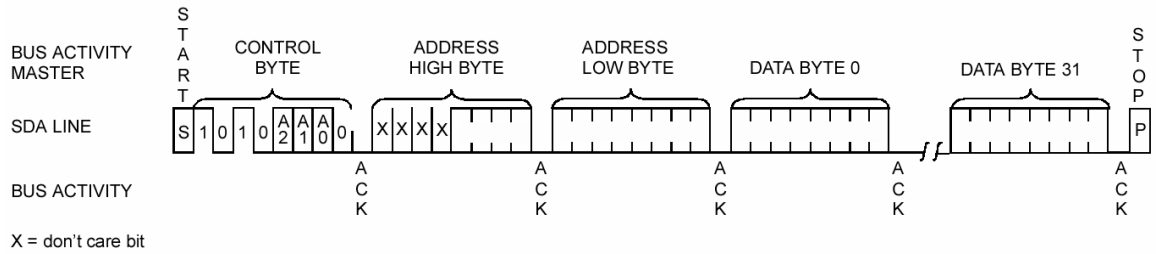


Figure 3.7 24LC32 datasheet excerpt on how to save a string of bytes to an address in the EEPROM.

Although this seems as straight-forward as the previous examples, it's not because the 24LC32's datasheet also mentions several rules that have to be followed by the I²C master (the Javelin Stamp) when writing more than one byte at a time. Specifically, you can only write up to 32 bytes at a time, and you have to wait after each 32-byte page is filled. Figure 3.8 (also from the 24LC32's data sheet) shows how to poll the chip to see if it's done transferring the 32 bytes to EEPROM or not. The 24LC32 does not reply with an ACK until it's done transferring data from its 32 byte buffer into the selected EEPROM addresses.

Digging deeper into the text in the 24LC32's datasheet, it turns out that you can't just arbitrarily write 32 bytes at a time. The entire 24LC32 memory is partitioned into 32-byte pages. You can write to addresses 0 – 31, but then you have to stop and poll until the EEPROM is ready again. Then, you can write to addresses 32 – 63 before you have to wait again. You could potentially write to addresses 50 – 63, but you still have to stop and poll before moving to address 64. The best way to figure out if you've reached a page boundary is to divide your EEPROM address by 32. If the remainder is zero, you know you have reached a page boundary. The modulus operator, **%**, can be used to give you the remainder of a division problem.

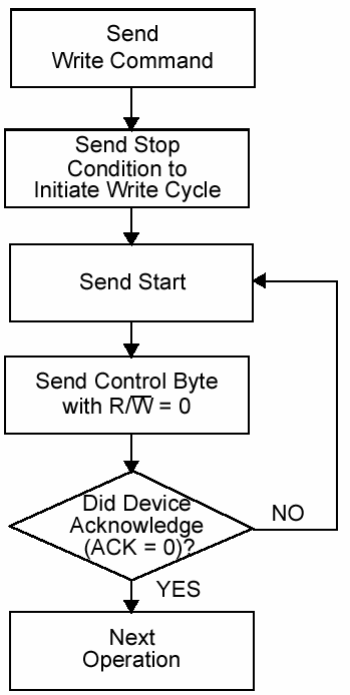


Figure 3.8
24LC32 datasheet excerpt on how to poll the a 24LC32 between each consecutive write to a 32-byte page.

Here is a code snippet that uses the modulus operator to stop when you reach an EEPROM page boundary and poll until the 24LC32 is ready for more bytes (Figure 3.8). Then it writes up to 32 more bytes (Figure 3.7).

```
// Set EEPROM address pointer.
int controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
i2cbus.start();
i2cbus.write(controlByte);
i2cbus.write(eeAddress >> 8);
i2cbus.write(eeAddress);

// Perform multiple writes, but stop before you cross a 32-byte
// page boundary and poll until an ACK is received.
for(int i = 0; i < characters.length(); i++){
  if(eeAddress %32){
    i2cbus.stop();
    do{
```

```
        i2cbus.start();
      } while(!i2cbus.write(controlByte));
      i2cbus.write(eeAddress >> 8);
      i2cbus.write(eeAddress);
    }
    i2cbus.write(characters.charAt(i));
    eeAddress++;
  }
  i2cbus.stop();
```

Example 4 - Reading a String

Figure 3.9 shows how to perform multiple reads from the 24LC32 to get a string of bytes.

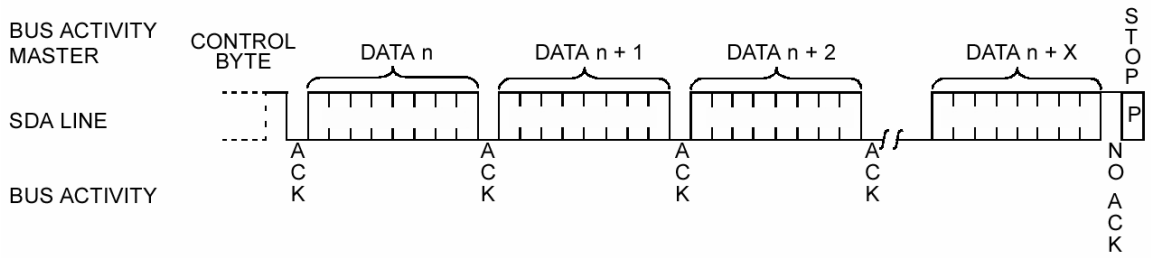


Figure 3.9 24LC32 datasheet excerpt on how to read a string of bytes.

The code below takes care of the I²C multi-read transactions shown in Figure 3.9.

```
controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | READ_BIT;
i2cbus.start();
i2cbus.write(controlByte);
for(int i = 0; i < numChars; i++){
  if(i < (numChars - 1)){
    characters.append((char)i2cbus.read(true));
  }else{
    characters.append((char)i2cbus.read(false));
}
```

The 24LC32 will return data starting at the most recent placement of its EEPROM address pointer. Below is the same code that precedes the single-byte read operation discussed in Example 2, and it can be used to set the starting EEPROM address for a multi-byte read operation.

```
int controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
```

```
i2cbus.start();
i2cbus.write(controlByte);
i2cbus.write(eeAddress >> 8);
i2cbus.write(eeAddress);
```

All Together Now

Program Listing 3.2 uses the code snippets from the four examples just discussed to create a tool that you can use to write data to and read data from the 24LC32. You can work with individual characters or strings.

If you experimented with

- ✓ Program Listing 3.1's response to different chip select wiring combinations, double check and make sure you re-wired the 24LC32 to hardware address 0 (A2, A1, and A0 tied to Vss).
- ✓ Open MC24LC32Demo.java from the Projects\examples\protocol\i2cprimer directory.
- ✓ Click the Javelin Stamp IDE's Run button.
- ✓ Follow the prompts in the Messages from Javelin Window (example shown in Figure 3.10).

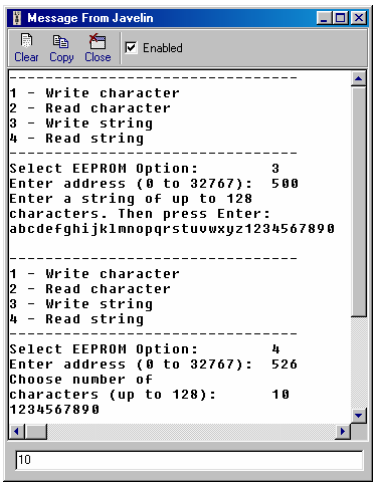


Figure 3.10 Messages From Javelin Window from Program Listing 3.2.

Remember to enter your replies to the prompts in the lower windowpane.

Program Listing 3.2 – Read and Write Characters and Strings

```
package examples.protocol.i2cprimer;

import stamp.core.*;
import stamp.protocol.I2C;

public class MC24LC32Demo {

  final public static int SDAPin = CPU.pin6;
```

```
final public static int SCLPin = CPU.pin7;
public static I2C i2cbus = new I2C(SDAPin, SCLPin);

final public static int READ_BIT      = 0x0001;
final public static int WRITE_BIT     = 0x0000;
final public static int CONTROL_CODE  = 0x00A0;
final public static int SLAVE_ADDRESS = 0x0000;

public static StringBuffer characters = new StringBuffer(128);
public static int menuChoice, eeAddress, data, controlByte, numChars;

public static void main() {

  while(true){

    menuChoice = TerminalHelper.menu(1,4);
    eeAddress = TerminalHelper.getAddress();

    switch (menuChoice){

      // Uses code discussed in Example-1 section.
      case '1':
        data = TerminalHelper.getCharacter();
        controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
        i2cbus.start();
        i2cbus.write(controlByte);
        i2cbus.write(eeAddress >> 8);
        i2cbus.write(eeAddress);
        i2cbus.write(data);
        i2cbus.stop();
        break;

      // Uses code discussed in Example-2 section.
      case '2':
        i2cbus.start();
        controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
        i2cbus.write(controlByte);
        i2cbus.write(eeAddress >> 8);
        i2cbus.write(eeAddress);
        controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | READ_BIT;
        i2cbus.start();
        i2cbus.write(controlByte);
        data = i2cbus.read(false);
        i2cbus.stop();
        TerminalHelper.announceCharacter((char)data);
        break;
```

```
      // Uses code discussed in Example-3 section.
      case '3':
        TerminalHelper.getString(characters);
        int controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
        i2cbus.start();
        i2cbus.write(controlByte);
        i2cbus.write(eeAddress >> 8);
        i2cbus.write(eeAddress);
        for(int i = 0; i < characters.length(); i++){
          if((eeAddress % 32) == 0 ){
            i2cbus.stop();
            do{
              i2cbus.start();
            } while(!i2cbus.write(controlByte));
            i2cbus.write(eeAddress >> 8);
            i2cbus.write(eeAddress);
          }
          i2cbus.write(characters.charAt(i));
          eeAddress ++;
        }
        i2cbus.stop();
        break;

      // Uses code discussed in Example-4 section.
      case '4':
        numChars = TerminalHelper.getCharCount();
        characters.clear();
        i2cbus.start();
        controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | WRITE_BIT;
        i2cbus.write(controlByte);
        i2cbus.write(eeAddress >> 8);
        i2cbus.write(eeAddress);
        controlByte = CONTROL_CODE | (SLAVE_ADDRESS << 1) | READ_BIT;
        i2cbus.start();
        i2cbus.write(controlByte);
        for(int i = 0; i < numChars; i++){
          if(i < (numChars - 1)){
            characters.append((char)i2cbus.read(true));
          }
          else{
            characters.append((char)i2cbus.read(false));
          }
        }
        TerminalHelper.displayCharacters(characters);
    }   // End switch
```

```
    }     // End while(true)
  }       // End main
}         // End class
```

Creating an I²C Device Library

Program Listing 3.2 and future programs that you write that use the 24LC32 (or an I²C device of your choice) can be greatly simplified by writing a library file for the device. There are many different ways to write a library file, and for the sake of simplicity, Library Listing 3.1 is just an example of how to convert the code from Program Listing 3.2 into a library file. A library file commonly contains a class definition, fields (constants and variables), constructors, methods, and lots of javadoc comments.

The class definition and field declarations are similar to what we've used in the previous two programs, but the constructor is new. Library Listing 3.1 was designed assuming that some other program or library file might be communicating with one or more I2C busses. Each new MC24LC32LibEx object needs to know which I²C bus has the 24LC32 chip connected to it, and it also needs to know the chip's hardware address on that bus. The constructor receives these two values and then assigns them to **this** instance of the MC24LC32 object.

```
public Microchip24LC32 (I2C i2cbus, int chipAddress) {
  this.i2cbus = i2cbus;
  this.deviceAddress = (CONTROL_CODE | (chipAddress << 1) & 0x00FF);
}
```

Tip	The MC24LC32LibEx library was written for the sake of demonstrating how a program that works with a particular device can be converted to a library file. A more full featured class that handles the 24LC32 (or possibly MC24LCXX series of devices) may have been written. If so, it would be available for download from the www.javelinstamp.com Applications page.

Each **case** statement in Program Listing 3.2 set the 24LC32's EEPROM address pointer. This code was intentionally redundant because it was written to match the diagrams from the 24LC32's data sheet. A single method named **setAddress** in the MC24LC32LibEx class was created for use by the other methods in the library.

```
private void setAddress(int eeAddress){
  int controlByte = deviceAddress | WRITE_BIT;
  i2cbus.start();
  i2cbus.write(controlByte);
  i2cbus.write(eeAddress >> 8);
  i2cbus.write(eeAddress);
}
```

The setAddress method is called by the other methods in the library to set the 24LC32's EEPROM address pointer before proceeding. For example, the **`readByte`** method, which was **`case: '2'`** in Program Listing 3.2, calls the **`setAddress`** method before performing the read. The **`readByte`** method expects to receive the address from the program that's requesting the byte and it returns the byte value it read from the 24LC32.

```
public int readByte(int eeAddress){
  setAddress(eeAddress);
  int controlByte = deviceAddress | READ_BIT;
  i2cbus.start();
  i2cbus.write(controlByte);
  int value = i2cbus.read(false);
  i2cbus.stop();
  return value;
}
```

An HTML file containing a brief set of instructions on how to use the library file can be automatically generated from comments placed between the **`/**`** and the **`*/`**. For example, the javadoc comment preceding the **`readByte`** method looks like this:

```
/**
 * Read a byte value from an address in the 24LC32.
 *
 * @param eeAddress the address that contains the byte to be read.
 * @return value the byte stored at <code>eeAddress</code>.
 */
```

These comments are referred to as javadoc comments, and they are generated by javadoc.exe in Sun Microsystems' Software Development Kit (SDK) available for free download from www.java.sun.com. You can see how these javadoc comments are converted into a web page by using your web browser to open MC24LC32LibEx.pdf from the \doc\ folder.

Library Listing 3.1 – Example Device Library for the 24LC32

```
package stamp.peripheral.memory.eeprom;

import stamp.core.*;
import stamp.protocol.I2C;

/**
 * This class is for demonstration purposes only.  It should be used in
 * conjunction with Javelin Stamp Application Note 3: I2C Primer - EEPROM
 * example.<p>
 *
```

```
* This class can be instantiated for each 24LC32 on a given I2C bus, and
* it contains methods that enable bytewise and multi-byte read and write
* operations.
* <p>
*
* @version 1.0 8 October 2002
* @author Andy Lindsay, Parallax Inc.
*/
public class MC24LC32LibEx {

 final private static int READ_BIT     = 0x0001;
 final private static int WRITE_BIT    = 0x0000;
 final private static int CONTROL_CODE = 0x00A0;

 private I2C i2cbus;
 private int deviceAddress, data;

 /**
  * Create MC24LC32 object by passing an I2C bus and the 24LC32's chip
  * address to this constructor.  For example:
  * <p><code>
  *  // Create an I2C bus object named i2cbus.
  *  final public static int SDAPin = CPU.pin6;
  *  final public static int SCLPin = CPU.pin7;
  *  public static I2C i2cbus = new I2C(SDAPin, SCLPin);
  *
  *  // Create a Microchip24LC32 object named eeprom0 using the i2cbus object.
  *  public static Microchip24LC32 eeprom0 = new Microchip24LC32(i2cbus, 0);
  * </code><p>
  * @param i2cbus the I2C bus object that has the new 24LC32 object/chip
  * connected to it.
  * @param chipAddress the binary address value of the new 24LC32 chip.
  * This should be the binary value of A2, A1, A0.
  */
 public MC24LC32LibEx (I2C i2cbus, int chipAddress) {
   this.i2cbus = i2cbus;
   this.deviceAddress = (CONTROL_CODE | (chipAddress << 1) & 0x00FF);
 }

 /**
  * Set the 24LC32's EEPROM address pointer.
  *
  * @param eeAddress
  */
 public void setAddress(int eeAddress){
   int controlByte = deviceAddress | WRITE_BIT;
   i2cbus.start();
```

```
  i2cbus.write(controlByte);
  i2cbus.write(eeAddress >> 8);
  i2cbus.write(eeAddress);
}

/**
 * Write a byte value to a particular address in the 24LC32.
 *
 * @param eeAddress the address where the byte value should be stored.
 * @param dataByte the byte value to be stored at <code>eeAddress</code>.
 */
public void writeByte(int eeAddress, int dataByte){
  setAddress(eeAddress);
  i2cbus.write(dataByte);
  i2cbus.stop();
}

/**
 * Read a byte value from an address in the 24LC32.
 *
 * @param eeAddress the address that contains the byte to be read.
 * @return value the byte stored at <code>eeAddress</code>.
 */
public int readByte(int eeAddress){
  setAddress(eeAddress);
  int controlByte = deviceAddress | READ_BIT;
  i2cbus.start();
  i2cbus.write(controlByte);
  int value = i2cbus.read(false);
  i2cbus.stop();
  return value;
}

/**
 * Write a string of characters starting at a particular address in
 * the 24LC32.
 *
 * @param eeAddress the address where the byte value should be stored.
 * @param sb the <code>StringBuffer</code> object that contains the string
 * of characters.
 */
public void writeStringToEeprom(int eeAddress, StringBuffer sb){
  setAddress(eeAddress);
  int controlByte = deviceAddress | WRITE_BIT;
  for(int i = 0; i < sb.length(); i++){
    if(eeAddress %32 == 0) {
      i2cbus.stop();
```

```
        do{
          i2cbus.start();
        } while(!i2cbus.write(controlByte));
        i2cbus.write(eeAddress >> 8);
        i2cbus.write(eeAddress);
      }
      i2cbus.write(sb.charAt(i));
      eeAddress ++;
    }
    i2cbus.stop();
  }

  /**
   * Read a string of characters of a specific length starting at a
   * paricular address in the 24LC32.
   *
   * @param eeAddress the starting address at the beginning of the string.
   * @param count the number of characters to read
   * @param sb the <code>StringBuffer</code> object that stores the string
   * of characters.
   */
  public void readStringIntoBuffer(int eeAddress, int count, StringBuffer sb){
    sb.clear();
    setAddress(eeAddress);
    int controlByte = deviceAddress | READ_BIT;
    i2cbus.start();
    i2cbus.write(controlByte);
    for(int i = 0; i < count; i++){
      if(i < (count - 1)){
        sb.append((char)i2cbus.read(true));
      }
      else{
        sb.append((char)i2cbus.read(false));
      }
    }
  }
}
```

Using an I²C Device Library

When you're using an the MC24LC32LibEx device library, you will need to import both the I2C library and the device library:

```
import stamp.protocol.I2C;
import stamp.peripheral.memory.eeprom.MC24LC32LibEx;
```

Next, inside the class definition, create an I2C bus object, and use it as a parameter to create the device object. This is how you tell the device which I²C bus it's on. This is especially important if you have more than one I²C bus with like devices on it.

```
// Create I2C object and pass it to the MC24LC32LibEx object.
public static I2C i2cbus = new I2C(SDAPin, SCLPin);
public static MC24LC32LibEx eeprom0 = new MC24LC32LibEx(i2cbus, 0);
```

Now you are ready to use the device object, named **eeprom0** in this case, to access the **MC24LC32LibEx** library's fields and methods. Since all the functions developed in Program Listing 3.2 have been converted to methods in the MC24LC32LibEx library class, the code becomes much simpler. Program Listing 3.3 makes use of the Messages from Javelin Window in the same way Program Listing 3.2 does, and its data storage and retrieval functionality is also the same.

Program Listing 3.3 – Example Code Simplified by the Device Library

```
package examples.protocol.i2cprimer;

import stamp.core.*;

// Import I2C and MC24LC32LibEx libraries.
import stamp.protocol.I2C;
import stamp.peripheral.memory.eeprom.MC24LC32LibEx;

public class MC24LC32LibExDemo {

  // Create an I2C bus object named i2cbus.
  final public static int SDAPin = CPU.pin6;
  final public static int SCLPin = CPU.pin7;
  public static I2C i2cbus = new I2C(SDAPin, SCLPin);

  // Create a MC24LC32LibExample object named eeprom0 using the i2cbus object.
  public static MC24LC32LibEx eeprom0 = new MC24LC32LibEx(i2cbus, 0);

  // Declare StringBuffer object and static variables.
  public static StringBuffer characters = new StringBuffer(128);
  public static int menuChoice, eeAddress, data, controlByte, numChars;

  public static void main() {

    while(true){

      // Get user input for menu choice and EEPROM address.
      menuChoice = TerminalHelper.menu(1,4);
      eeAddress = TerminalHelper.getAddress();
```

```
      // Each case interacts with the user using the TerminalHelper object
      // and reads/writes the 24LC32 using eeprom0, an MC24LC32LibEx object.
      switch (menuChoice){

        case '1':
          data = TerminalHelper.getCharacter();
          eeprom0.writeByte(eeAddress, data);
          break;

        case '2':
          data = eeprom0.readByte(eeAddress);
          TerminalHelper.announceCharacter((char)data);
          break;

        case '3':
          TerminalHelper.getString(characters);
          eeprom0.writeStringToEeprom(eeAddress,characters);
          break;

        case '4':
          numChars = TerminalHelper.getCharCount();
          characters.clear();
          eeprom0.readStringIntoBuffer(eeAddress, numChars, characters);
          TerminalHelper.displayCharacters(characters);
      }   // End switch
    }     // End while(true)
  }       // End main
}         // End class
```

Published Resources – for More Information

Not only are I²C and I²C compatible devices versatile and plentiful, there is a wealth of useful design and application information available for free download from the web. Below are the documents used to write this application note.

"I²C *Bus Specification Version 2.1"*, Phillips Semiconductor, January, 2000.

The I²C bus specification are available for free download from www.philipslogic.com. Phillips also has a number of MS Power Point presentations in PDF format available for free download from their site. They are worthwhile for reviewing to get a better idea of how to use I²C peripherals in your project.

"24AA32/24LC32 32 k I²C™ Serial EEPROM", Data Sheet, Microchip Corporation, 2002.

The 24LC32 data sheet is available for free download from www.microchip.com, and it was useful for developing the example code.

"Javelin Stamp Manual", Users Manual, Version 1.0, Parallax, Inc., 2002

The Javelin Stamp Manual from the www.javelinstamp.com Documents page, contains more information about the techniques used in this application note's example programs and the library files.

"BASIC Stamp Manual", Users Manual, Version 2.0c, Parallax, Inc., 2000

The BASIC Stamp Manual from the www.parallaxinc.com Downloads → Documentation page, contains more information about the I²C protocol with PBASIC examples for the BASIC Stamp 2p.

Javelin Stamp Discussion Forum – Questions and Answers

The Parallax, Inc. Javelin Stamp Discussion Forum is a searchable repository of questions and answers about the Javelin Stamp and Javelin Stamp Applications. To view the Javelin Stamp Forum, go to www.javelinstamp.com and follow the Discussion link. You can also join this forum and post your own questions. Other Javelin Stamp users who monitor the list will see your questions and reply with helpful tips, part numbers, pointers to useful web pages, etc.