Syllabus:

EMBEDDED COMMUNICATION PROTOCOLS: Embedded Networking: Introduction – Serial/Parallel Communication – Serial communication protocols -RS232 standard – RS485 – Synchronous Serial Protocols -Serial Peripheral Interface (SPI) – Inter Integrated Circuits (I2C) – PC Parallel port programming -ISA/PCI Bus protocols – Firewire.

USB Bus: Introduction – Speed Identification on the bus – USB States – USB bus communication: Packets –Data flow types –Enumeration –Descriptors –PIC 18 Microcontroller USB Interface

CAN Bus: Introduction - Frames –Bit stuffing –Types of errors –Nominal Bit Timing – PIC microcontroller CAN Interface –A simple application with CAN.

EMBEDDED ETHERNET: Exchanging messages using UDP and TCP – Serving web pages with Dynamic Data – Serving web pages that respond to user Input – Email for Embedded Systems – Using FTP – Keeping Devices and Network secure.

WIRELESS EMBEDDED NETWORKING: Wireless sensor networks – Introduction – Applications – Network Topology – Localization –Time Synchronization - Energy efficient MAC protocols –SMAC – Energy efficient and robust routing – Data Centric routing

TEXT BOOKS

1. Frank Vahid, Givargis 'Embedded Systems Design: A Unified Hardware/Software Introduction', Wiley Publications
2. Jan Axelson, 'Parallel Port Complete', Penram publications
3. Dogan Ibrahim, 'Advanced PIC microcontroller projects in C', Elsevier 2008
4. Jan Axelson 'Embedded Ethernet and Internet Complete', Penram publications
5. Bhaskar Krishnamachari, 'Networking wireless sensors', Cambridge press 2005

USB Bus: Introduction
Speed Identification on the bus
USB States
USB bus communication: Packets
Data flow types
Enumeration
Descriptors
PIC 18 Microcontroller USB Interface

CHAPTER 8

Advanced PIC18 Projects—USB Bus Projects

The Universal Serial Bus (USB) is one of the most common interfaces used in electronic consumer products today, including PCs, cameras, GPS devices, MP3 players, modems, printers, and scanners, to name a few.

The USB was originally developed by Compaq, Microsoft, Intel, and NEC, and later by Hewlett-Packard, Lucent, and Philips as well. These companies eventually formed the nonprofit corporation USB Implementers Forum Inc. to organize the development and publication of USB specifications.

This chapter describes the basic principles of the USB bus and shows how to use USB-based applications with PIC microcontrollers. The USB bus is a complex protocol. A complete discussion of its design and use is beyond the scope of this chapter. Only the basic principles, enough to be able to use the USB bus, are outlined here. On the other hand, the functions offered by the mikroC language that simplify the design of USB-based microcontroller projects are described in some detail.

The USB is a high-speed serial interface that can also provide power to devices connected to it. A USB bus supports up to 127 devices (limited by the 7-bit address field—note that address 0 is not used as it has a special purpose) connected through a four-wire serial cable of up to three or even five meters in length. Many USB devices can be connected to the same bus with hubs, which can have 4, 8, or even 16 ports. A device can be plugged into a hub which is plugged into another hub, and so on. The maximum number of tiers permitted is six. According to the specification, the maximum distance of a device from its host is about thirty meters, accomplished by using five hubs. For longer-distance bus communications, other methods such as use of Ethernet are recommended.

The USB bus specification comes in two versions: the earlier version, USB1.1, supports 11Mbps, while the new version, USB 2.0, supports up to 480Mbps. The USB specification defines three data speeds:

- Low speed—1.5Mb/sec
- Full speed—12Mb/sec
- High speed—480Mb/sec

The maximum power available to an external device is limited to about 100mA at 5.0V.

USB is a four-wire interface implemented using a four-core shielded cable. Two types of connectors are specified and used: Type A and Type B. Figure 8.1 shows typical USB connectors. Figure 8.2 shows the pin-out of the USB connectors.

The signal wire colors are specified. The pins and wire colors of a Type A or Type B connector are given in Table 8.1.

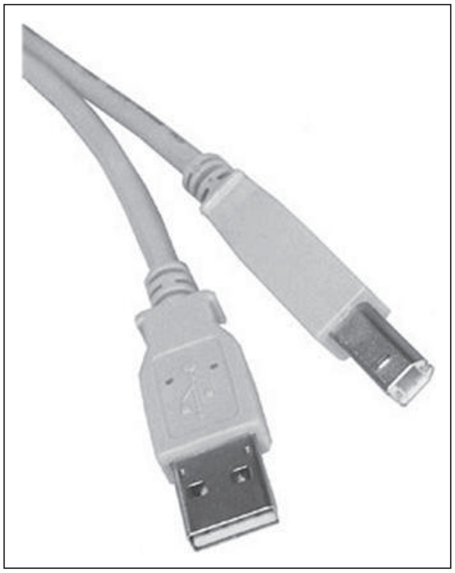


Figure 8.1: USB connectors

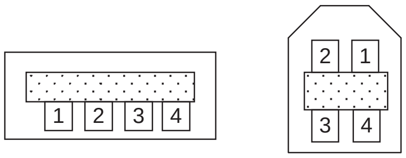


Figure 8.2: Pin-out of USB connectors

Table 8.1: USB connector pin assignments

Pin no.	Name	Color
1	+5.0V	Red
2	Data−	White
3	Data+	Green
4	Ground	Black

The specification also defines a mini-B connector, mainly used in smaller portable electronic devices such as cameras and other handheld devices. This connector has a fifth pin called ID, though this pin is not used. The pin assignment and wire colors of a mini-B connector are given in Table 8.2.

Two of the pins, Data+ and Data−, form a twisted pair and carry differential data signals and some single-ended data states.

Table 8.2: Mini USB pin assignments

Pin no.	Name	Color
1	+5.0V	Red
2	−Data	White
3	+Data	Green
4	Not used	–
5	Ground	Black

USB signals are bi-phase, and signals are sent from the host computer using the NRZI (non-return to zero inverted) data encoding technique. In this technique, the signal level is inverted for each change to a logic 0. The signal level for a logic 1 is not changed. A 0 bit is “stuffed” after every six consecutive ones in the data stream to make the data dynamic (this is called *bit stuffing* because the extra bit lengthens the data stream). Figure 8.3 shows how the NRZI is implemented.

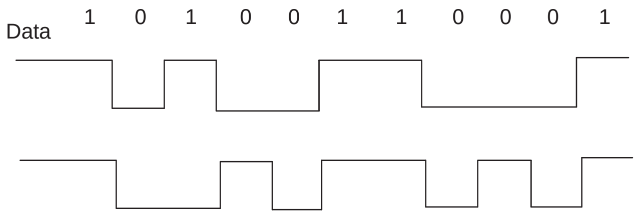


Figure 8.3: NRZI data

A packet of data transmitted by the host is sent to every device connected to the bus, traveling downward through the chain of hubs. All the devices receive the signal, but only one of them, the addressed one, accepts the data. Conversely, only one device at any time can transmit to the host, and the data travels upward through the chain of hubs until it reaches the host.

USB devices attached to the bus may be full-custom devices, requiring a full-custom device driver, or they may belong to a device class. Device classes enable the same device driver to be used for several devices having similar functionalities. For example, a printer device has the device class 0×07, and most printers use drivers of this type.

The most common device classes are given in Table 8.3. The USB human interface device (HID) class is of particular interest, as it is used in the projects in this chapter.

Some common USB terms are:

Endpoint: An endpoint is either a source or a sink of data. A single USB device can have a number of endpoints, the limit being sixteen IN and sixteen OUT endpoints.

Transaction: A transaction is a transfer of data on the bus.

Pipe: A pipe is a logical data connection between the host and an endpoint.

Table 8.3: USB device classes

Device class	Description	Example device
0×00	Reserved	–
0×01	USB audio device	Sound card
0×02	USB communications device	Modem, fax
0×03	USB human interface device	Keyboard, mouse
0×07	USB printer device	Printer
0×08	USB mass storage device	Memory card, flash drive
0×09	USB hub device	Hubs
0×0B	USB smart card reader device	Card reader
0×0E	USB video device	Webcam, scanner
0×E0	USB wireless device	Bluetooth

8.1 Speed Identification on the Bus

At the device end of the bus, a 1.5K pull-up resistor is connected from the D+ or D– line to 3.3V. On a full-speed bus, the resistor is connected from the D+ line to 3.3V, and on a low-speed bus the resistor is from D– line to 3.3V. When no device is plugged in, the host will see both data lines as low. Connecting a device to the bus will pull either the D+ or the D– line to logic high, and the host will know that a device is plugged into the bus. The speed of the device is determined by observing which line is pulled high.

8.2 USB States

Some of the USB bus states are:

Idle: The bus is in idle state when the pulled-up line is high and the other line is low. This is the state of the lines before and after a packet transmission.

Detached: When no device is connected to the bus, the host sees both lines as low.

Attached: When a device is connected to the bus, the host sees either D+ or D– go to logic high, which means a device has been plugged in.

J state: The same as idle state.

K state: The opposite of J state.

SE0: The single ended zero state, where both lines on the bus are pulled low.

SE1: The single ended one state, where both lines on the bus are high. SE1 is an illegal condition on the bus; it must never be in this state.

Reset: When the host wants to communicate with a device on the bus, it first sends a "reset" condition by pulling low both data lines (SE0 state) for at least 10ms.

EOP: The end of packet state, which is basically an SE0 state for 2 bit times, followed by a J state for 1 bit time.

Keep alive: The state achieved by EOP. Keep alive is sent at least once every millisecond to keep the device from suspending.

Suspend: Used to save power, suspend is implemented by not sending anything to a device for 3ms. A suspended device draws less than 0.5mA from the bus and must recognize reset and resume signals.

Resume: A suspended device is woken up by reversing the polarity of the signal on the data lines for at least 20ms, followed by a low-speed EOP signal.

8.3 USB Bus Communication

USB is a host-centric connectivity system where the host dictates the use of the USB bus. Each device on the bus is assigned a unique USB address, and no slave device can assert a signal on the bus until the host asks for it. When a new USB device is plugged into a bus, the USB host uses address 0 to ask basic information from the device. Then the host assigns it a unique USB address. After the host asks for and receives further information about the device, such as the name of the manufacturer, device capabilities, and product ID, two-way transactions on the bus can begin.

8.3.1 Packets

Data is transmitted on a USB bus in packets. A packet starts with a sync pattern to allow the receiver clock to synchronize with the data. The data bytes of the packet follow, ending with an end of packet signal.

A packet identifier (PID) byte immediately follows the sync field of every USB packet. A PID itself is 4 bits long, and the 4 bits are repeated in a complemented form. There are seventeen different PID values, as shown in Table 8.4. These include one reserved value and one that is used twice, with two different meanings.

There are four packet formats, based on which PID is at the start of the packet: token packets, data packets, handshake packets, and special packets.

Figure 8.4 shows the format of a token packet, which is used for OUT, IN, SOF (start of frame), and SETUP. The packet contains a 7-bit address, a 4-bit ENDP (endpoint number), a 5-bit CRC checksum, and an EOP (end of packet).

Table 8.4: PID values

PID type	PID name	Bits	Description
Token	OUT IN SOF SETUP	1110 0001 0110 1001 1010 0101 0010 1101	Host to device transaction Device to host transaction Start of frame Setup command
Data	DATA0 DATA1 DATA2 MDATA	1100 0011 0100 1011 1000 0111 0000 1111	Data packet PID even Data packet PID odd Data packet PID high speed Data packet PID high speed
Handshake	ACK NAK STALL NYET	1101 0010 0101 1010 0001 1110 1001 0110	Receiver accepts packet Receiver does not accept packet Stalled No response from receiver
Special	PRE ERR SPLIT PING Reserved	0011 1100 0011 1100 0111 1000 1011 0100 1111 0000	Host preample Split transaction error High-speed split transaction High-speed flow control Reserved

Sync	PID	ADDR	ENDP	CRC	EOP
	8 bits	7 bits	4 bits	5 bits	

Figure 8.4: Token packet

A data packet is used for DATA0, DATA1, DATA2, and MDATA data transactions. The packet format is shown in Figure 8.5 and consists of the PID, 0–1024 bytes of data, a 2-byte CRC checksum, and an EOP.

Sync	PID	Data	CRC	EOP
	1 byte	0–1024 bytes	2 bytes	

Figure 8.5: Data packet

Sync	PID	EOP
	1 byte	

Figure 8.6: Handshake packet

Figure 8.6 shows the format of a handshake packet, which is used for ACK, NAK, STALL, and NYET. ACK is used when a receiver acknowledges that it has received an error-free data packet. NAK is used when the receiving device cannot accept the packet. STALL indicates when the endpoint is halted, and NYET is used when there is no response from the receiver.

8.3.2 Data Flow Types

Data can be transferred on a USB bus in four ways: bulk transfer, interrupt transfer, isochronous transfer, and control transfer.

Bulk transfers are designed to transfer large amounts of data with error-free delivery and no guarantee of bandwidth. If an OUT endpoint is defined as using bulk transfers, then the host will transfer data to it using OUT transactions. Similarly, if an IN endpoint is defined as using bulk transfers, then the host will transfer data from it using IN transactions. In general, bulk transfers are used where a slow rate of transfer is not a problem. The maximum packet size in a bulk transfer is 8 to 64 packets at full speed, and 512 packets at high speed (bulk transfers are not allowed at low speeds).

Interrupt transfers are used to transfer small amounts of data with a high bandwidth where the data must be transferred as quickly as possible with no delay. Note that interrupt transfers have nothing to do with interrupts in computer systems. Interrupt packets can range in size from 1 to 8 bytes at low speed, from 1 to 64 bytes at full speed, and up to 1024 bytes at high speed.

Isochronous transfers have a guaranteed bandwidth, but error-free delivery is not guaranteed. This type of transfer is generally used in applications, such as audio data transfer, where speed is important but the loss or corruption of some data is not. An isochronous packet may contain 1023 bytes at full speed or up to 1024 bytes at high speed (isochronous transfers are not allowed at low speeds).

A *control transfer* is a bidirectional data transfer, using both IN and OUT endpoints. Control transfers are generally used for initial configuration of a device by the host. The maximum packet size is 8 bytes at low speed, 8 to 64 bytes at full speed, and 64 bytes at high speed. A control transfer is carried out in three stages: SETUP, DATA, and STATUS.

8.3.3 Enumeration

When a device is plugged into a USB bus, it becomes known to the host through a process called enumeration. The steps of enumeration are:

- When a device is plugged in, the host becomes aware of it because one of the data lines (D+ or D−) becomes logic high.
- The host sends a USB reset signal to the device to place the device in a known state. The reset device responds to address 0.
- The host sends a request on address 0 to the device to find out its maximum packet size using a *Get Descriptor* command.
- The device responds by sending a small portion of the device descriptor.
- The host sends a USB reset again.
- The host assigns a unique address to the device and sends a *Set Address* request to the device. After the request is completed, the device assumes the new address. At this point the host is free to reset any other newly plugged-in devices on the bus.
- The host sends a *Get Device Descriptor* request to retrieve the complete device descriptor, gathering information such as manufacturer, type of device, and maximum control packet size.
- The host sends a *Get Configuration Descriptors* request to receive the device's configuration data, such as power requirements and the types and number of interfaces supported.
- The host may request any additional descriptors from the device.

The initial communication between the host and the device is carried out using the control transfer type of data flow.

Initially, the device is addressed, but it is in an unconfigured state. After the host gathers enough information about the device, it loads a suitable device driver which configures the device by sending it a *Set Configuration* request. At this point the device has been configured, and it is ready to respond to device-specific requests (i.e., it can receive data from and send data to the host).

8.4 Descriptors

All USB devices have a hierarchy of descriptors that describe various features of the device: the manufacturer ID, the version of the device, the version of USB it supports, what the device is, its power requirements, the number and type of endpoints, and so forth.

The most common USB descriptors are:

- Device descriptors
- Configuration descriptors
- Interface descriptors
- HID descriptors
- Endpoint descriptors

The descriptors are in a hierarchical structure as shown in Figure 8.7. At the top of the hierarchy we have the device descriptor, then the configuration descriptors, followed by the interface descriptors, and finally the endpoint descriptors. The HID descriptor always follows the interface descriptor when the interface belongs to the HID class.

All descriptors have a common format. The first byte (*bLength*) specifies the length of the descriptor, while the second byte (*bDescriptorType*) indicates the descriptor type.

8.4.1 Device Descriptors

The device descriptor is the top-level set of information read from a device and the first item the host attempts to retrieve.

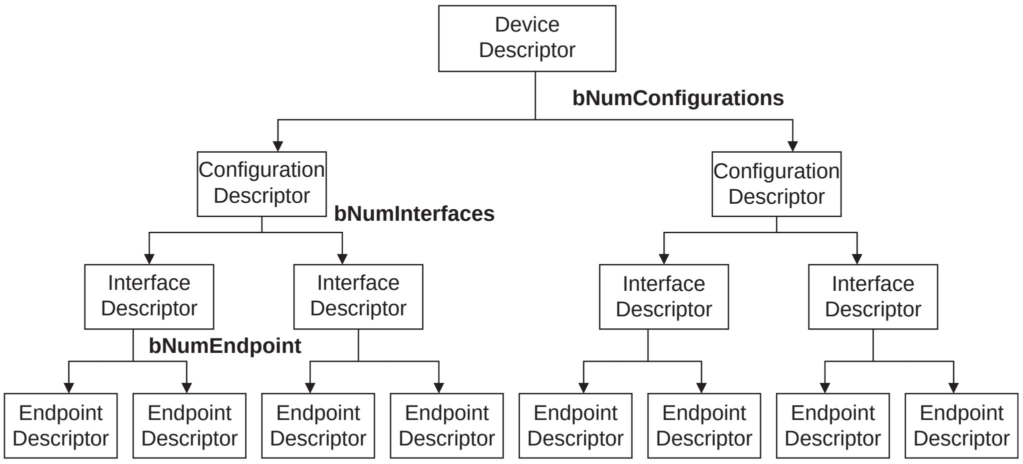


Figure 8.7: USB descriptor hierarchy

A USB device has only one device descriptor, since the device descriptor represents the entire device. It provides general information such as manufacturer, serial number, product number, the class of the device, and the number of configurations. Table 8.5 shows the format for a device descriptor with the meaning of each field.

bLength is the length of the device descriptor.

bDescriptorType is the descriptor type.

bcdUSB reports the highest version of USB the device supports in BCD format. The number is represented as 0×JJMN, where JJ is the major version number, M is the minor version number, and N is the subminor version number. For example, USB 1.1 is reported as 0×0110.

bDeviceClass, *bDeviceSubClass*, and *bDeviceProtocol* are assigned by the USB organization and are used by the system to find a class driver for the device.

bMaxPacketSize0 is the maximum input and output packet size for endpoint 0.

idVendor is assigned by the USB organization and is the vendor's ID.

idProduct is assigned by the manufacturer and is the product ID.

bcdDevice is the device release number and has the same format as the *bcdUSB*.

Table 8.5: Device descriptor

Offset	Field	Size	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	Device descriptor (0×01)
2	bcdUSB	2	Highest version of USB supported
4	bDeviceClass	1	Class code
5	bDeviceSubClass	1	Subclass code
6	bDeviceProtocol	1	Protocol code
7	bMaxPacketSize0	1	Maximum packet size
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Device release number
14	iManufacturer	1	Manufacturer string descriptor
15	iProduct	1	Index of product string descriptor
16	iSerialNumber	1	Index of serial number descriptor
17	bNumConfigurations	1	Number of possible configurations

iManufacturer, *iProduct*, and *iSerialNumber* are details about the manufacturer and the product. These fields have no requirement and can be set to zero.

bNumConfigurations is the number of configurations the device supports.

Table 8.6 shows an example device descriptor for a mouse device. The length of the descriptor is 18 bytes (*bLength* = 18), and the descriptor type is 0×01 (*bDescriptorType* = 0×01). The device supports USB 1.1 (*bcdUSB* = 0×0110). *bDeviceClass*, *bDeviceSubClass*, and *bDeviceProtocol* are set to zero to show that the class information is in the interface descriptor. *bMaxPacketSize0* is set to 8 to show that the maximum input and output packet size for endpoint 0 is 8 bytes. The next three bytes identify the device by the vendor ID, product ID, and device version number. The next three items define indexes to strings about the manufacturer, product, and the serial number. Finally, we notice that the mouse device has just one configuration (*bNumConfigurations* = 1).

Table 8.6: Example device descriptor

Offset	Field	Value	Description
0	bLength	18	Size is 18
1	bDescriptorType	0×01	Descriptor type
2	bcdUSB	0×0110	Highest USB supported = USB 1.1
4	bDeviceClass	0×00	Class information in interface descriptor
5	bDeviceSubClass	0×00	Class information in interface descriptor
6	bDeviceProtocol	0×00	Class information in interface descriptor
7	bMaxPacketSize0	8	Maximum packet size
8	idVendor	0×02A	XYZ Co Ltd.
10	idProduct	0×1001	Mouse
12	bcdDevice	0×0011	Device release number
14	iManufacturer	0×20	Index to manufacturer string
15	iProduct	0×21	Index of product string
16	iSerialNumber	0×22	Index of serial number string
17	bNumConfigurations	1	Number of possible configurations

8.4.2 Configuration Descriptors

The configuration descriptor provides information about the power requirements of the device and how many different interfaces it supports. There may be more than one configuration for a device.

Table 8.7 shows the format of the configuration descriptor with the meaning of each field.

bLength is the length of the device descriptor.

bDescriptorType is the descriptor type.

wTotalLength is the total combined size of this set of descriptors (i.e., total of configuration descriptor + interface descriptor + HID descriptor + endpoint descriptor). When the configuration descriptor is read by the host, it returns the entire configuration information, which includes all interface and endpoint descriptors.

Table 8.7: Configuration descriptor

Offset	Field	Size	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	Device descriptor (0×02)
2	wTotalLength	2	Total bytes returned
4	bNumInterfaces	1	Number of interfaces
5	bConfigurationValue	1	Value used to select configuration
6	iConfiguration	1	Index describing configuration string
7	bmAttributes	1	Power supply attributes
8	bMaxPower	2	Max power consumption in 2mA

bNumInterfaces is the number of interfaces present for this configuration.

bConfigurationValue is used by the host (in command *SetConfiguration*) to select the configuration.

iConfiguration is an index to a string descriptor describing the configuration in readable format.

bmAttributes describes the power requirements of the device. If the device is USB bus-powered, then bit D7 is set. If it is self-powered, it sets bit D6. Bit D5 specifies the remote wakeup of the device. Bits D7 and D0–D4 are reserved.

bMaxPower defines the maximum power the device will draw from the bus in 2mA units.

Table 8.8 shows an example configuration descriptor for a mouse device. The length of the descriptor is 9 bytes (*bLength* = 9), and the descriptor type is 0×02 (*bDescriptorType* = 0×02). The total combined size of the descriptors is 34 (*wTotalLength* = 34). The number of interfaces for the mouse device is 1 (*bNumInterfaces* = 1). *Host SetConfiguration* command must use the value 1 as an argument in *SetConfiguration()* to select this configuration. There is no string to describe this configuration. *bmAttributes* is set to 0×40 to indicate that the device is self-powered. *bMaxPower* is set to 10 to specify that the maximum current drawn by the device is 20mA.

Table 8.8: Example configuration descriptor

Offset	Field	Value	Description
0	bLength	9	Descriptor size is 9 bytes
1	bDescriptorType	0×02	Device descriptor is 0×02
2	wTotalLength	34	Total bytes returned is 34
4	bNumInterfaces	1	Number of interfaces is 1
5	bConfigurationValue	1	Value used to select configuration
6	iConfiguration	0×2A	Index describing configuration string
7	bmAttributes	0×40	Power supply attributes
8	bMaxPower	10	Max power consumption is 20mA

8.4.3 Interface Descriptors

The interface descriptors specify the class of the interface and the number of endpoints it uses. There may be more than one interface.

Table 8.9 shows the format of the interface descriptor with the meaning of each field.

Table 8.9: Interface descriptor

Offset	Field	Size	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	Device descriptor (0×04)
2	bInterfaceNumber	1	Number of interface
3	bAlternateSetting	1	Value to select alternate setting
4	bNumEndpoints	1	Number of endpoints
5	bInterfaceClass	1	Class code
6	bInterfaceSubClass	1	Subclass code
7	bInterfaceProtocol	1	Protocol code
8	iInterface	1	Index of string descriptor to interface

bLength is the length of the device descriptor.

bDescriptorType is the descriptor type.

bInterfaceNumber indicates the index of the interface descriptor.

bAlternateSetting can be used to specify alternate interfaces that can be selected by the host using command *Set Interface*.

bNumEndpoints indicates the number of endpoints used by the interface.

bInterfaceClass specifies the device class code (assigned by the USB organization).

bInterfaceSubClass specifies the device subclass code (assigned by the USB organization).

bInterfaceProtocol specifies the device protocol code (assigned by the USB organization).

iInterface is an index to a string descriptor of the interface.

Table 8.10 shows an example interface descriptor for a mouse device. The descriptor length is 9 bytes (*bLength* = 9) and the descriptor type is 0×04 (*bDescriptorType* = 0×04). The interface number used to reference this interface is 1 (*bInterfaceNumber* = 1).

Table 8.10: Example interface descriptor

Offset	Field	Value	Description
0	bLength	9	Descriptor size is 9 bytes
1	bDescriptorType	0×04	Device descriptor is 0×04
2	bInterfaceNumber	0	Number of interface
3	bAlternateSetting	0	Value to select alternate setting
4	bNumEndpoints	1	Number of endpoints is 1
5	bInterfaceClass	0×03	Class code is 0×03
6	bInterfaceSubClass	0×02	Subclass code is 0×02
7	bInterfaceProtocol	0×02	Protocol code is 0×02
8	iInterface	0	Index of string descriptor to interface

bAlternateSetting is set to 0 (i.e., no alternate interfaces). The number of endpoints used by this interface is 1 (excluding endpoint 0), and this is the endpoint used for the mouse to send its data. The device class code is 0×03 (*bInterfaceClass* = 0×03). This is an HID (human interface device) type class. The interface subclass is set to 0×02. The device protocol is 0×02 (mouse). There is no string to describe this interface (*iInterface* = 0).

8.4.4 HID Descriptors

An HID descriptor always follows an interface descriptor when the interface belongs to the HID class. Table 8.11 shows the format of the HID descriptor.

bLength is the length of the device descriptor.

bDescriptorType is the descriptor type.

bcdHID is the HID class specification.

bCountryCode specifies any special local changes.

bNumDescriptors specifes if there are any additional descriptors associated with this class.

bDescriptorType is the type of the additional descriptor specified in *bNumDescriptors*.

wDescriptorLength is the length of the additional descriptor in bytes.

Table 8.11: HID descriptor

Offset	Field	Size	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	HID (0×21)
2	bcdHID	2	HID class
4	bCountryCode	1	Special country dependent code
5	bNumDescriptors	1	Number of additional descriptors
6	bDescriptorType	1	Type of additional descriptor
7	wDescriptorLength	2	Length of additional descriptor

Table 8.12 shows an example HID descriptor for a mouse device. The length of the descriptor is 9 bytes (*bLength* = 9), and the descriptor type is 0×21 (*bDescriptorType* = 0×21). The HID class is set to 1.1 (*bcdHID* = 0×0110). The country code is set to zero (*bCountryCode* = 0), specifying that there is no special localization with this device. The number of descriptors is set to 1 (*bNumDescriptors* = 1) which specifies that there is one additional descriptor associated with this class. The type of the additional descriptor is REPORT (*bDescriptorType* = REPORT), and its length is 52 bytes (*wDescriptorLength* = 52).

Table 8.12: Example HID descriptor

Offset	Field	Value	Description
0	bLength	9	Descriptor size is 9 bytes
1	bDescriptorType	0×21	HID (0×21)
2	bcdHID	0×0110	Class version 1.1
4	bCountryCode	0	No special country dependent code
5	bNumDescriptors	1	Number of additional descriptors
6	bDescriptorType	REPORT	Type of additional descriptor
7	wDescriptorLength	5	Length of additional descriptor

8.4.5 Endpoint Descriptors

Table 8.13 shows the format of the endpoint descriptor.

bLength is the length of the device descriptor.

bDescriptorType is the descriptor type.

bEndpointAddress is the address of the endpoint.

bmAttributes specifies what type of endpoint it is.

wMaxPacketSize is the maximum packet size.

bInterval specifies how often the endpoint should be polled (in ms).

Table 8.14 shows an example endpoint descriptor for a mouse device. The length of the descriptor is 7 bytes (*bLength* = 7), and the descriptor type is 0×05 (*bDescriptorType*

Table 8.13: Endpoint descriptor

Offset	Field	Size	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	Endpoint (0×05)
2	bcdEndpointAddress	1	Endpoint address
4	bmAttributes	1	Type of endpoint
5	wMaxPacketSize	2	Max packet size
6	bInterval	1	Polling interval

Table 8.14: Example endpoint descriptor

Offset	Field	Size	Description
0	bLength	7	Descriptor size in bytes
1	bDescriptorType	0×05	Endpoint (0×05)
2	bcdEndpointAddress	0×50	Endpoint address
4	bmAttributes	0×03	Interrupt type endpoint
5	wMaxPacketSize	0×0002	Max packet size is 2
6	bInterval	0×14	Polling interval is 20ms

= 0×05). The endpoint address is 0×50 (*bEndpointAddress* = 0×50). The endpoint is to be used as an interrupt endpoint (*bmAttributes* = 0×03). The maximum packet size is set to 2 (*wMaxPacketSize* = 0×02) to indicate that packets longer than 2 bytes will not be sent from the endpoint. The endpoint should be polled at least once every 20ms (*bInterval* = 0×14).

8.5 PIC18 Microcontroller USB Bus Interface

Some of the PIC18 microcontrollers support USB interface directly. For example, the PIC18F4550 microcontroller contains a full-speed and low-speed compatible USB interface that allows communication between a host PC and the microcontroller. In the USB projects in this chapter we will use the PIC18F4550 microcontroller.

Figure 8.8 is an overview of the USB section of the PIC18F4550 microcontroller. PORTC pins RC4 (pin 23) and RC5 (pin 24) are used for USB interface. RC4 is the USB data D− pin, and RC5 is the USB data D+ pin. Internal pull-up resistors are provided which can be disabled (setting *UPUEN* = 0) if desired and external pull-up resistors can be used instead. For full-speed operation an internal or external resistor should be connected to data pin D+, and for low-speed operation an internal or external resistor should be connected to data pin D−.

Operation of the USB module is configured using three control registers, and a total of twenty-two registers are used to manage the actual USB transactions. Configuration

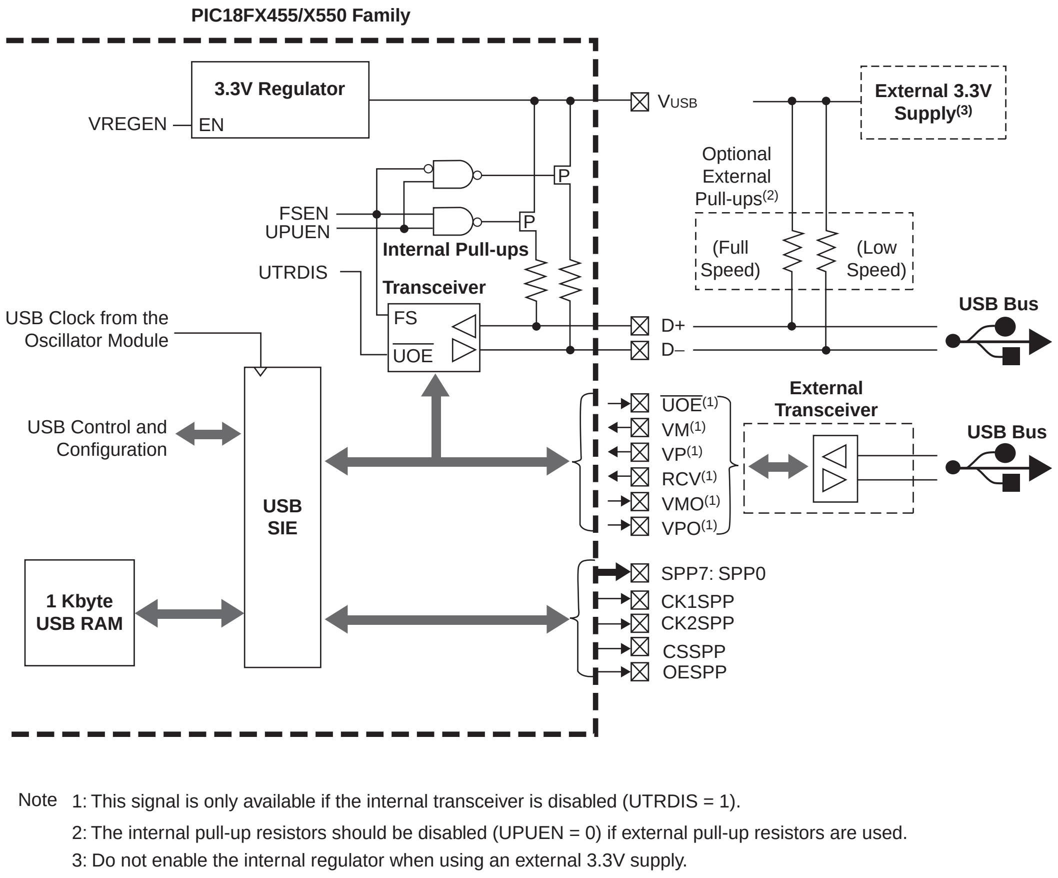

PIC18FX455/X550 Family
3.3V Regulator
EN
VREGEN
VUSB
External 3.3V Supply(3)
Optional External Pull-ups(2)
(Full Speed)
(Low Speed)
FSEN
UPUEN
P
P
Internal Pull-ups
UTRDIS
Transceiver
FS
UOE
D+
D–
USB Bus
USB Clock from the Oscillator Module
USB Control and Configuration
USB SIE
1 Kbyte USB RAM
UOE(1)
VM(1)
VP(1)
RCV(1)
VMO(1)
VPO(1)
External Transceiver
USB Bus
SPP7: SPP0
CK1SPP
CK2SPP
CSSPP
OESPP
Note 1: This signal is only available if the internal transceiver is disabled (UTRDIS = 1).
2: The internal pull-up resistors should be disabled (UPUEN = 0) if external pull-up resistors are used.
3: Do not enable the internal regulator when using an external 3.3V supply.

Figure 8.8: PIC18F4550 microcontroller USB overview

of these registers is a highly complex task and is not covered in this book. Interested readers should refer to the PIC18F4550 data sheet and to books on USB internals. In this chapter we are using the mikroC language USB library functions to implement USB transactions. The details of these functions are given in the next section.

8.6 mikroC Language USB Bus Library Functions

The mikroC language supports a number of functions for USB HID-type communications. Each project based on the USB library should include a descriptor source file which contains vendor ID and name, product ID and name, report length, and other relevant information. To create a descriptor source file we can use mikroC's integrated USB HID terminal tool (see *Tools → HID Terminal*). The default name for descriptor file is *USBdsc.c*, but it can be renamed if required. The *USBdsc.c* file must be included in USB-based projects either via the mikroC IDE tool, or as an *#include* option in the program source file.

The mikroC language supports the following USB bus library functions when a PIC microcontroller with built-in USB is used (e.g., PIC18F4550), and port pins RC4 and RC5 are connected to the D+ and D− pins of the USB connector respectively:

Hid_Enable: This function enables USB communication and requires two arguments: the read-buffer address and the write-buffer address. It must be called before any other functions of the USB library, and it returns no data.

Hid_Read: This function receives data from the USB bus and stores it in the receive-buffer. It has no arguments but returns the number of characters received.

Hid_Write: This function sends data from the write-buffer to the USB bus. The name of the buffer (the same buffer used in the initialization) and the length of the data to be sent must be specified as arguments to the function. The function does not return any data.

Hid_Disable: This function disables the USB data transfer. It has no arguments and returns no data.

The USB interface of a PIC18F4550 microcontroller is shown in Figure 8.9. As the figure shows, the interface is very simple. In addition to the power supply and ground pins, it requires just two pins to be connected to the USB connector. The microcontroller receives power from the USB port.

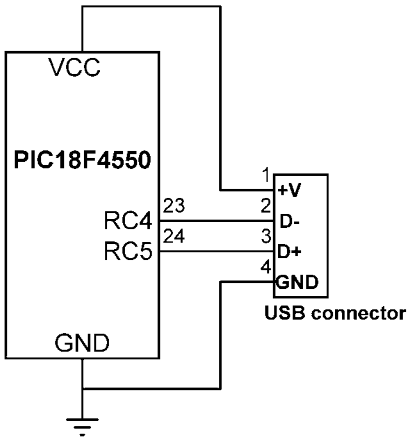


Figure 8.9: PIC18F4550 USB interface

PROJECT 8.1—USB-Based Microcontroller Output Port

This project describes the design of a USB-based microcontroller output port. A PIC18F4550 microcontroller is interfaced to a PC through a USB cable. A Visual Basic program runs on the PC and sends commands to the microcontroller through the USB bus, asking the microcontroller to set/reset the I/O bits of its PORTB.

The block diagram of the project is shown in Figure 8.10. The circuit diagram is given in Figure 8.11. The USB lines of the PIC18F4550 microcontroller are connected to a USB connector. The microcontroller is powered from the USB line (i.e., no external

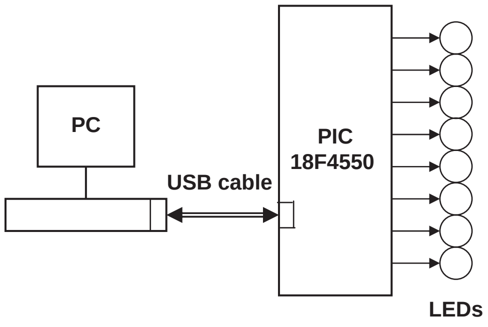


Figure 8.10: Block diagram of the project

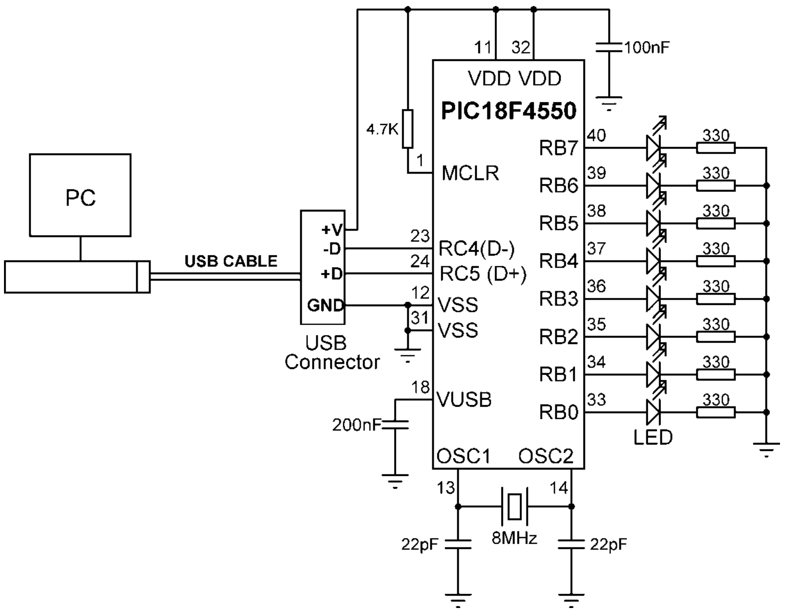


Figure 8.11: Circuit diagram of the project

power supply is required). This makes the design of USB-based products relatively cheap and very attractive in applications where the total power consumption is below 100mA. The microcontroller is operated from an 8MHz crystal.

The PORTB pins of the microcontroller are connected to LEDs so we can see the state changes as commands are sent from the PC. This makes testing the project very easy. Note that a capacitor (about 200nF) should be connected between the V_{USB} pin (pin 18) of the microcontroller and the ground for stability.

The project software consists of two parts: the PC software, and the microcontroller software. Both are described in this section.

The PC Software

The PC software is based on Visual Basic. It is assumed that the user has elementary knowledge of Visual Basic programming language. Instruction in programming using the Visual Basic language is beyond the scope of this book, and interested readers should refer to various books available on this topic.

The source program listing and the executables of the programs are given on the CDROM distributed with this book. Readers who do not want to do any programming can use or modify the given programs.

The Visual Basic program in this example consists of a single form as shown in Figure 8.12. The required PORTB data should be entered in decimal in the text box, and then the command button CLICK TO SEND should be clicked with the mouse. For example, entering decimal number 15 will turn on the LEDs connected to port pins RB0,RB1,RB2, and RB3 of PORTB.

The program sends the entered number to the microcontroller as a packet consisting of four characters in the following format:

$$P = nT$$

where character P indicates the start of data, n is the byte to be sent to PORTB, and T is the terminator character.

For example, if bits 3 and 4 of PORTB are to be set, i.e., PORTB = "00011000," then the Visual Basic program sends packet P = 24T (number 24 is sent as a single binary byte and not as two ASCII bytes) to the microcontroller over the USB link. The bottom part of the form displays the connection status.

The Visual Basic program used in this section is based on the USB utility known as EasyHID USB Wizard, developed by Mecanique, and can be downloaded free of charge

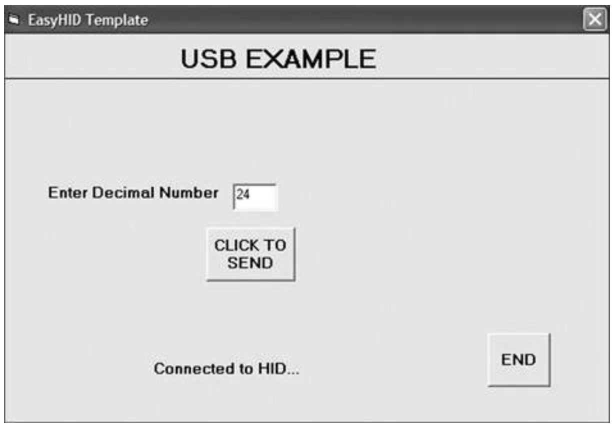
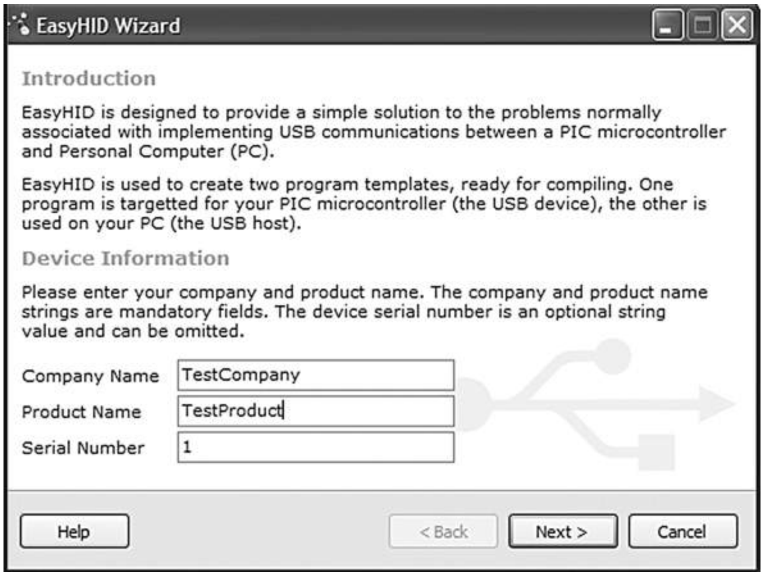


Figure 8.12: The PC Visual Basic form

from their web site (www.mecanique.co.uk). EasyHID is designed to work with USB 2.0, and there is no need to develop a driver, as the XP operating system is shipped with a HID-based USB driver. This utility generates Visual Basic, Visual C++, or Borland Delphi template codes for the PC end of a USB application using an HID-type device interface. In addition, the utility can generate USB template code for the PIC18F4550 and similar microcontrollers, based on the Proton Development Suite (www.crownhill.co.uk), Swordish PIC Basic, or PicBasic Pro (www.melabs.com) programming languages. The generated codes can be expanded with the user code to implement the required application.

The steps in generating a Visual Basic code template follow:

- Load the EasyHID zip file from the Mecanique web site by clicking on "Download EasyHID as a Standalone Application"
- Extract the files and install the application by double-clicking on SETUP.
- When the program has started, you should see a form as shown in Figure 8.13. Enter your data in the fields Company Name, Product Name, and the optional Serial Number.



Introduction

EasyHID is designed to provide a simple solution to the problems normally associated with implementing USB communications between a PIC microcontroller and Personal Computer (PC).

EasyHID is used to create two program templates, ready for compiling. One program is targetted for your PIC microcontroller (the USB device), the other is used on your PC (the USB host).

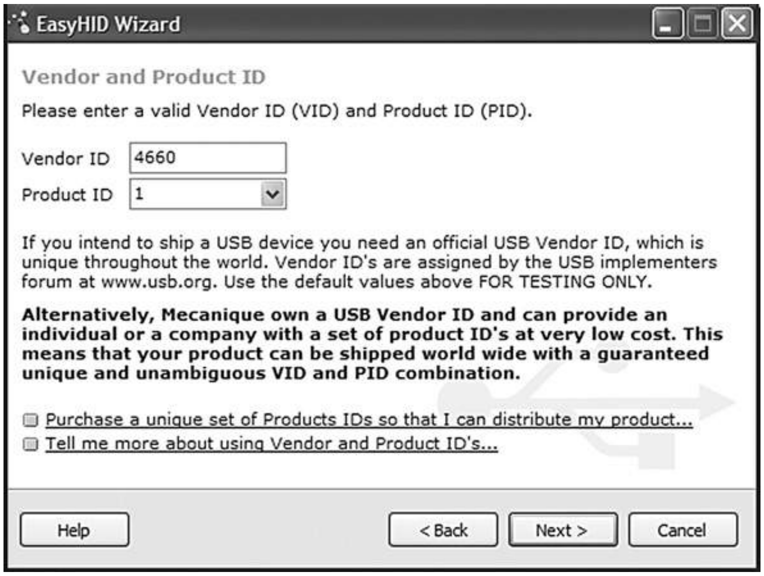
Device Information

Please enter your company and product name. The company and product name strings are mandatory fields. The device serial number is an optional string value and can be omitted.

Company Name	TestCompany
Product Name	TestProduct
Serial Number	1

Figure 8.13: EasyHID first form

- Enter your Vendor ID (VID) and Product ID (PID) as shown in the form in Figure 8.14. Vendor IDs are unique throughout the world and are issued by the USB implementers (www.usb.org) at a cost. Mecanique owns a Vendor ID and can issue you a set of Product IDs at low cost so your products can be shipped all over the world with unique VID and PID combinations. In this example, VID = 4660 and PID = 1 are selected for test purposes.



Vendor and Product ID

Please enter a valid Vendor ID (VID) and Product ID (PID).

Vendor ID 4660

Product ID 1

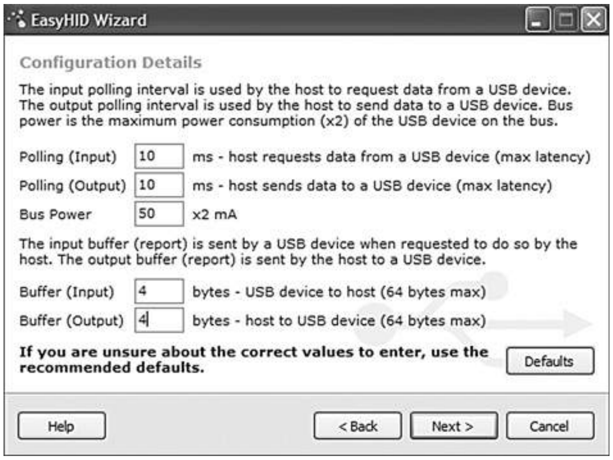
If you intend to ship a USB device you need an official USB Vendor ID, which is unique throughout the world. Vendor ID's are assigned by the USB implementers forum at www.usb.org. Use the default values above FOR TESTING ONLY.

Alternatively, Mecanique own a USB Vendor ID and can provide an individual or a company with a set of product ID's at very low cost. This means that your product can be shipped world wide with a guaranteed unique and unambiguous VID and PID combination.

☐ Purchase a unique set of Products IDs so that I can distribute my product...
☐ Tell me more about using Vendor and Product ID's...

Figure 8.14: EasyHID VID and PID entry form

- Clicking *Next* displays the form shown in Figure 8.15. The important parameters here are the output and input buffer sizes, which specify the number of bytes to be sent and received respectively between the PC and the microcontroller during USB data transactions. In this example, 4 bytes are chosen for both fields (our output is in the format P = nT, which is 4 bytes).
- In the next form (see Figure 8.16), select a location for the generated files, choose the microcontroller compiler to be used (this field is not important, as we are only generating code for Visual Basic (i.e., the PC



Configuration Details

The input polling interval is used by the host to request data from a USB device. The output polling interval is used by the host to send data to a USB device. Bus power is the maximum power consumption (x2) of the USB device on the bus.

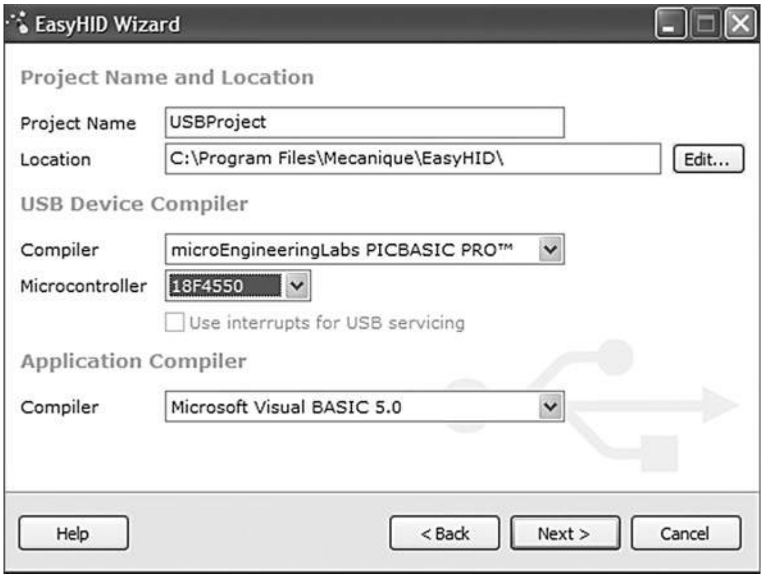
Polling (Input)	10	ms - host requests data from a USB device (max latency)
Polling (Output)	10	ms - host sends data to a USB device (max latency)
Bus Power	50	x2 mA

The input buffer (report) is sent by a USB device when requested to do so by the host. The output buffer (report) is sent by the host to a USB device.

Buffer (Input)	4	bytes - USB device to host (64 bytes max)
Buffer (Output)	4	bytes - host to USB device (64 bytes max)

If you are unsure about the correct values to enter, use the recommended defaults.

Figure 8.15: EasyHID input-output buffer selection



Project Name and Location

Project Name: USBProject
Location: C:\Program Files\Mecanique\EasyHID\

USB Device Compiler

Compiler: microEngineeringLabs PICBASIC PRO™
Microcontroller: 18F4550
Use interrupts for USB servicing

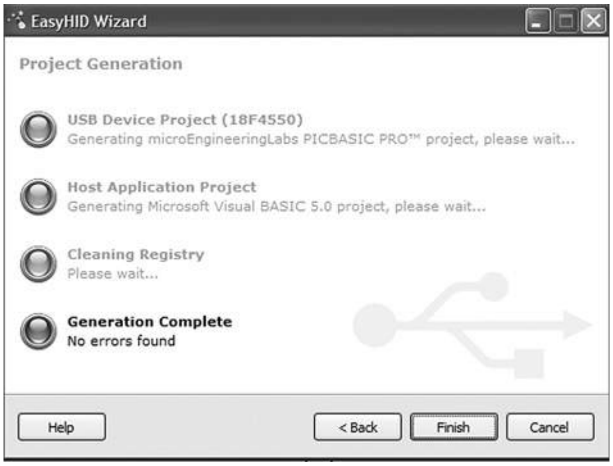
Application Compiler

Compiler: Microsoft Visual BASIC 5.0

Figure 8.16: EasyHID output folder, microcontroller type, and host compiler selection

end), choose the microcontroller type, and finally select Visual Basic as the language to be used.

- Clicking *Next* generates Visual Basic and microcontroller code templates in the selected directories (see the final form in Figure 8.17).



USB Device Project (18F4550)
Generating microEngineeringLabs PICBASIC PRO™ project, please wait...

Host Application Project
Generating Microsoft Visual BASIC 5.0 project, please wait...

Cleaning Registry
Please wait...

Generation Complete
No errors found

Figure 8.17: EasyHID last form

Figure 8.18 shows the Visual Basic files generated by the EasyHID wizard. The files basically consist of a blank form (FormMain.frm), a module file (mcHIDInterface. BAS), and a project file (USBProject.vbp).

The files generated by the EasyHID wizard have been modified for our project as follows:

- The blank form has been modified to display the various controls shown in Figure 8.12.
- Messages are added to the program to display when a USB device is plugged into or unplugged from the PC.
- A subroutine has been added to read the data entered by the user and then send this data to the microcontroller over the USB bus when the button CLICK TO SEND is clicked. This code is as follows:

```
Private Sub Command2_Click()

    BufferOut(0) = 0            ' first by is always the report ID
    BufferOut(1) = Asc("P")     ' first data item ("P")
    BufferOut(2) = Asc("=")     ' second data item ("=")
    BufferOut(3) = Val(txtno)   ' third data item (number to send)
    BufferOut(4) = Asc("T")     ' fourth data item ("T")

    ' write the data (don't forget, pass the whole array)...
    hidWriteEx VendorID, ProductID, BufferOut(0)
    lblstatus = "Data sent..."
End Sub
```

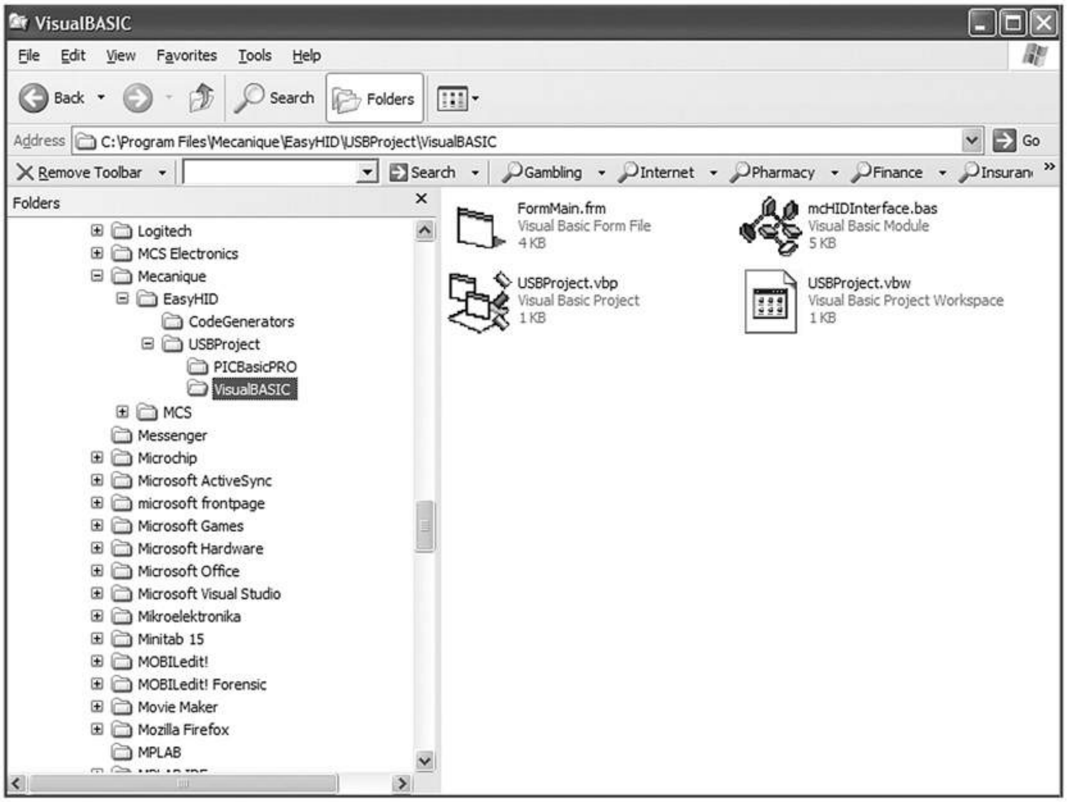


Figure 8.18: Files generated by the EasyHID wizard

BufferOut stores the data to be sent to the microcontroller over the USB bus. Notice that the first byte of this buffer is the report ID and must be set to 0. The actual data starts from address *BufferOut(1)* of the array and the data sent is in the format P = nT as described before. After the data is sent, the message "Data sent..." appears at the bottom part of the display.

Figure 8.19 shows the final listing of the Visual Basic program. The program is in two parts: the form USB1.FRM and the module USB1.BAS. The programs should be loaded and used in the Visual Basic development environment. An installable version of this program (in folder USB1) comes with the CDROM included with this book for those who do not have the Visual Basic development environment. This program should be installed as a normal Windows software installation.

The Microcontroller Software

The microcontroller receives the command *P = nT* from the PC and sends data byte *n* to PORTB. The listing of the microcontroller program (USB.C) without the USB code is shown in Figure 8.20. The program configures PORTB as digital output.

Generating the USB Descriptor File

The USB descriptor file must be included at the beginning of the mikroC program. This descriptor file is created using the Tools menu option of the mikroC compiler as follows:

- Select *Tools -> HID Terminal*
- A new form should be displayed. Click on the Descriptor tab and the form shown in Figure 8.21 is displayed.
- The important parameters to enter here are vendor ID (VID), product ID (PID), input buffer size, output buffer size, vendor name (VN), and product name (PN). Note that the VID and PID are in hexadecimal format and that the values entered here must be the same as the ones used in the Visual Basic program when generating the code using the EasyHID wizard. Choose VID = 1234 (equivalent to decimal 6460), PID = 1, input buffer size = 4, output buffer size = 4, and any names you like for the VN and PN fields.
- Check the mikroC compiler.

```
USB1.FRM

' vendor and product IDs
Private Const VendorID = 4660
Private Const ProductID = 1

' read and write buffers
Private Const BufferInSize = 8
Private Const BufferOutSize = 8
Dim BufferIn(0 To BufferInSize) As Byte
Dim BufferOut(0 To BufferOutSize) As Byte

Private Sub Command1_Click()
        Form_Unload (0)
        End
End Sub

Private Sub Command2_Click()
        BufferOut(0) = 0                    ' first by is always the report ID
        BufferOut(1) = Asc("P")             ' first data item (“P”)
        BufferOut(2) = Asc("=")             ' second data item (“-“)
        BufferOut(3) = Val(txtno)           ' third data item (to send over USB)
        BufferOut(4) = Asc("T")             ' fourth data item (“T”)

        ' write the data (don't forget, pass the whole array)...
        hidWriteEx VendorID, ProductID, BufferOut(0)
        lblstatus = "Data sent..."
End Sub

' ****************************************************************************
' when the form loads, connect to the HID controller - pass
' the form window handle so that you can receive notification
' events...
'*****************************************************************************
Private Sub Form_Load()
  ' do not remove!
        ConnectToHID (Me.hwnd)
        lblstatus = "Connected to HID..."
End Sub

'*****************************************************************************
' disconnect from the HID controller...
'*****************************************************************************
Private Sub Form_Unload(Cancel As Integer)
        DisconnectFromHID
End Sub


'*****************************************************************************
' a HID device has been plugged in...
```

Figure 8.19: Visual Basic program for the PC end of USB link

```
'*******************************************************************************
Public Sub OnPlugged(ByVal pHandle As Long)
  If hidGetVendorID(pHandle) = VendorID And hidGetProductID(pHandle)=
ProductID Then
        lblstatus = "USB Plugged....."
        End If
End Sub

'*******************************************************************************

' a HID device has been unplugged...
'*******************************************************************************
Public Sub OnUnplugged(ByVal pHandle As Long)
  If hidGetVendorID(pHandle) = VendorID And hidGetProductID(pHandle) =
ProductID Then
        lblstatus = "USB Unplugged...."
  End If
End Sub

'*******************************************************************************

' controller changed notification - called
' after ALL HID devices are plugged or unplugged
'*******************************************************************************
Public Sub OnChanged()
  Dim DeviceHandle As Long

        ' get the handle of the device we are interested in, then set
        ' its read notify flag to true - this ensures you get a read
        ' notification message when there is some data to read...
        DeviceHandle = hidGetHandle(VendorID, ProductID)
        hidSetReadNotify DeviceHandle, True
End Sub

'*******************************************************************************

' on read event...
'*******************************************************************************
Public Sub OnRead(ByVal pHandle As Long)

  ' read the data (don't forget, pass the whole array)...
  If hidRead(pHandle, BufferIn(0)) Then
    ' ** YOUR CODE HERE **
    ' first byte is the report ID, e.g. BufferIn(0)
    ' the other bytes are the data from the microcontrolller...
  End If
End Sub
```

Figure 8.19: (Cont'd)

USB1.BAS

```
' this is the interface to the HID controller DLL - you should not
' normally need to change anything in this file.
'
' WinProc() calls your main form 'event' procedures - these are currently
' set to..
'
' MainForm.OnPlugged(ByVal pHandle as long)
' MainForm.OnUnplugged(ByVal pHandle as long)
' MainForm.OnChanged()
' MainForm.OnRead(ByVal pHandle as long)

Option Explicit

' HID interface API declarations...
Declare Function hidConnect Lib "mcHID.dll" Alias "Connect" (ByVal pHostWin As
Long) As Boolean
Declare Function hidDisconnect Lib "mcHID.dll" Alias "Disconnect" () As Boolean
Declare Function hidGetItem Lib "mcHID.dll" Alias "GetItem" (ByVal pIndex As
Long) As Long
Declare Function hidGetItemCount Lib "mcHID.dll" Alias "GetItemCount" () As
Long
Declare Function hidRead Lib "mcHID.dll" Alias "Read" (ByVal pHandle As Long,
ByRef pData As Byte) As Boolean
Declare Function hidWrite Lib "mcHID.dll" Alias "Write" (ByVal pHandle As Long,
ByRef pData As Byte) As Boolean
Declare Function hidReadEx Lib "mcHID.dll" Alias "ReadEx" (ByVal pVendorID As
Long, ByVal pProductID As Long, ByRef pData As Byte) As Boolean
Declare Function hidWriteEx Lib "mcHID.dll" Alias "WriteEx" (ByVal pVendorID
As Long, ByVal pProductID As Long, ByRef pData As Byte) As Boolean
Declare Function hidGetHandle Lib "mcHID.dll" Alias "GetHandle" (ByVal
pVendoID As Long, ByVal pProductID As Long) As Long
Declare Function hidGetVendorID Lib "mcHID.dll" Alias "GetVendorID" (ByVal
pHandle As Long) As Long
Declare Function hidGetProductID Lib "mcHID.dll" Alias "GetProductID" (ByVal
pHandle As Long) As Long
Declare Function hidGetVersion Lib "mcHID.dll" Alias "GetVersion" (ByVal
pHandle As Long) As Long
Declare Function hidGetVendorName Lib "mcHID.dll" Alias "GetVendorName"
(ByVal pHandle As Long, ByVal pText As String, ByVal pLen As Long) As Long
Declare Function hidGetProductName Lib "mcHID.dll" Alias "GetProductName"
(ByVal pHandle As Long, ByVal pText As String, ByVal pLen As Long) As Long
Declare Function hidGetSerialNumber Lib "mcHID.dll" Alias"GetSerialNumber"
(ByVal pHandle As Long, ByVal pText As String, ByVal pLen As Long) As Long
Declare Function hidGetInputReportLength Lib "mcHID.dll" Alias
"GetInputReportLength" (ByVal pHandle As Long) As Long
Declare Function hidGetOutputReportLength Lib "mcHID.dll" Alias
"GetOutputReportLength" (ByVal pHandle As Long) As Long
```

Figure 8.19: (Cont'd)

```
Declare Sub hidSetReadNotify Lib "mcHID.dll" Alias "SetReadNotify" (ByVal
pHandle As Long, ByVal pValue As Boolean)
Declare Function hidIsReadNotifyEnabled Lib "mcHID.dll" Alias
"IsReadNotifyEnabled" (ByVal pHandle As Long) As Boolean
Declare Function hidIsAvailable Lib "mcHID.dll" Alias "IsAvailable" (ByVal
pVendorID As Long, ByVal pProductID As Long) As Boolean

' windows API declarations - used to set up messaging...
Private Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA"
(ByVal lpPrevWndFunc As Long, ByVal hwnd As Long, ByVal Msg As Long,
ByVal wParam As Long, ByVal lParam As Long) As Long
Private Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA"
(ByVal hwnd As Long, ByVal nIndex As Long, ByVal dwNewLong As Long) As
Long

' windows API Constants
Private Const WM_APP = 32768
Private Const GWL_WNDPROC = -4

' HID message constants
Private Const WM_HID_EVENT = WM_APP + 200
Private Const NOTIFY_PLUGGED = 1
Private Const NOTIFY_UNPLUGGED = 2
Private Const NOTIFY_CHANGED = 3
Private Const NOTIFY_READ = 4

' local variables
Private FPrevWinProc As Long    ' Handle to previous window procedure
Private FWinHandle As Long      ' Handle to message window

' Set up a windows hook to receive notification
' messages from the HID controller DLL - then connect
' to the controller
Public Function ConnectToHID(ByVal pHostWin As Long) As Boolean
  FWinHandle = pHostWin
  ConnectToHID = hidConnect(FWinHandle)
  FPrevWinProc = SetWindowLong(FWinHandle, GWL_WNDPROC, AddressOf
WinProc)
End Function

' Unhook from the HID controller and disconnect...
Public Function DisconnectFromHID() As Boolean
  DisconnectFromHID = hidDisconnect
  SetWindowLong FWinHandle, GWL_WNDPROC, FPrevWinProc
End Function

' This is the procedure that intercepts the HID controller messages...
Private Function WinProc(ByVal pHWnd As Long, ByVal pMsg As Long,
ByVal wParam As Long, ByVal lParam As Long) As Long
  If pMsg = WM_HID_EVENT Then
```

Figure 8.19: (Cont'd)

```
    Select Case wParam

     ' HID device has been plugged message...
     Case Is = NOTIFY_PLUGGED
       MainForm.OnPlugged (lParam)

     ' HID device has been unplugged
     Case Is = NOTIFY_UNPLUGGED
       MainForm.OnUnplugged (lParam)

     ' controller has changed...
     Case Is = NOTIFY_CHANGED
       MainForm.OnChanged

     ' read event...
     Case Is = NOTIFY_READ
       MainForm.OnRead (lParam)
     End Select

  End If

  ' next...
  WinProc = CallWindowProc(FPrevWinProc, pHWnd, pMsg, wParam, lParam)

End Function
```

Figure 8.19: (Cont'd)

- Clicking the CREATE button will ask for a folder name and then create descriptor file USBdsc in this folder. Rename this file to have extension ".C" (i.e., the full file name should be USBdsc.C) and then copy it to the following folder (other required mikroC files are already in this folder, so it makes sense to copy USBdsc.C here as well).

  ```
  C:\Program Files\Mikroelektronika\mikroC\Examples\EasyPic4
  \extra_examples\HID-library\USBdsc.c
  ```

Do not modify the contents of file USBdsc.C. A listing of this file is given on the CDROM.

The microcontroller program listing with the USB code included is shown in Figure 8.22 (program USB1.C). At the beginning of the program the USB descriptor file USBdsc.C is included. The operation of the USB link requires the microcontroller to keep the connection alive by sending keep-alive messages to the PC every several milliseconds. This is achieved by setting up a timer interrupt service routine using

```
/*****************************************************************************
                    USB BASED MICROCONTROLLER OUTPUT PORT
                    ========================================

In this project a PIC18F4550 type microcontroller is connected to a PC through
the USB link.

A Visual Basic program runs on the PC where the user enters the bits to be set
or cleared on PORTB of the microcontroller. The PC sends a command to the
microcontroller requesting it to set or reset the required bits of the microcontroller
PORTB.

The command sent by the PC to the microcontroller is in the following format:

         P=nT

where n is the byte the microcontroller is requested to send to PORTB of the
microcontroller.

Author:        Dogan Ibrahim
Date:          September 2007
File:          USB.C
*****************************************************************************/

void main()
{
    ADCON1 = 0xFF;                              // Set PORTB to digital I/O
    TRISB = 0;                                  // Set PORTB to outputs
    PORTB = 0;                                  // Clear all outputs
}
```

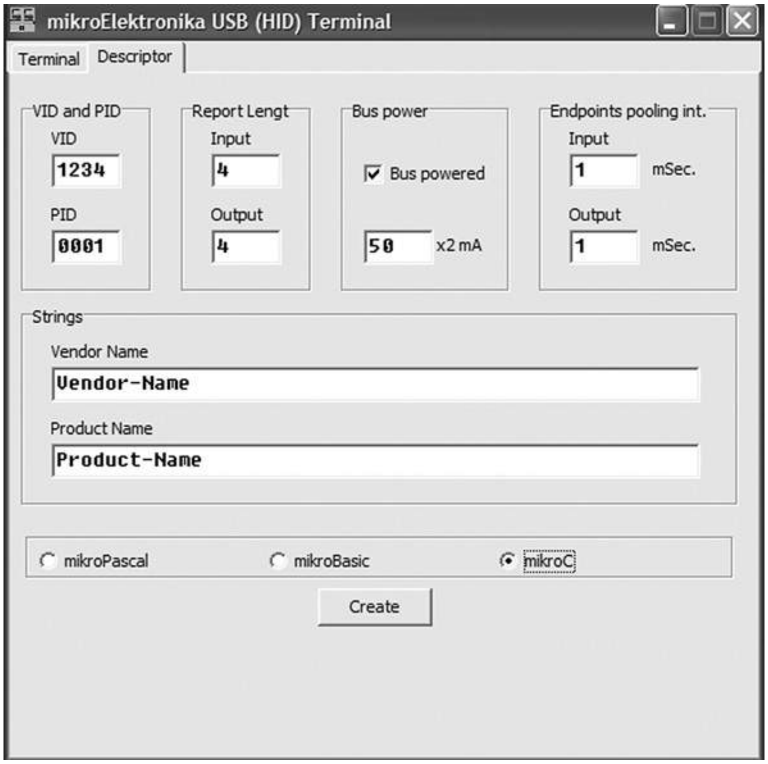
Figure 8.20: Microcontroller program without the USB code

TIMER 0. Inside the timer interrupt service routine the mikroC USB function *HID_InterruptProc* is called. Timer TMR0L is reloaded and timer interrupts are re-enabled just before returning from the interrupt service routine.

Inside the main program PORTB is defined as digital I/O and TRISB is cleared to 0 so all PORTB pins are outputs. All the interrupt registers are then set to their power-on-reset values for safety. The timer interrupts are then set up. The timer is operated in 8-bit mode with a prescaler of 256. Although the crystal clock frequency is 8MHZ, the CPU is operated with a 48MHz clock, as described later. Selecting a timer value of TMR0L = 100 with a 48MHz clock (CPU clock period of 0.083μs) gives timer interrupt intervals of:

$$(256 - 100) * 256 * 0.083\mu s$$

or, about 3.3ms. Thus, the keep-alive messages are sent every 3.3ms.


mikroElektronika USB (HID) Terminal

Terminal | Descriptor

VID and PID		Report Lengt		Bus power		Endpoints pooling int.	
VID	1234	Input	4	[✓] Bus powered		Input	1 mSec.
PID	0001	Output	4	50 x2 mA		Output	1 mSec.

Strings

Vendor Name: Vendor-Name

Product Name: Product-Name

mikroPascal [mikroBasic [✓ mikroC]

Figure 8.21: Creating the USBdsc descriptor file

The USB port is then enabled by calling function *Hid_Enable*. The program then enters an indefinite loop and reads data from the USB port with *Hid_Read*. When 4 bytes are received at the correct format (i.e., byte 0 = "P," byte 1 = "=", and byte 3 = "T") then the data byte is read from byte 2 and sent to PORTB of the microcontroller.

It is important to note that when data is received using the *Hid_Read* function, the function returns the number of bytes received. In addition, the first byte received is the first actual data byte and not the report ID.

Microcontroller Clock

The USB module of the PIC18F4550 microcontroller requires a 48MHz clock.
In addition, the microcontroller CPU requires a clock that can range from 0 to 48MHz.
In this project the CPU clock is set to be 48MHz.

There are several ways to provide the required clock pulses.

```
/******************************************************************************
                  USB BASED MICROCONTROLLER OUTPUT PORT
                  ========================================

In this project a PIC18F4550 type microcontroller is connected
to a PC through the USB link.

A Visual Basic program runs on the PC where the user enters the bits to be set or
cleared on PORTB of the microcontroller. The PC sends a command to the
microcontroller requesting it to set or reset the required bits of the microcontroller
PORTB.

A 8MHz crystal is used to operate the microcontroller. The actual CPU clock is raised
to 48MHz by setting configuration bits. Also, the USB module is operated with
48MHz.

The command sent by the PC to the microcontroller is in the following format:

        P=nT

where n is the byte the microcontroller is requested to send to PORTB of the
microcontroller.

This program includes the USB code.

Author:         Dogan Ibrahim
Date:           September 2007
File:           USB1.C
******************************************************************************/

#include "C:\Program
Files\Mikroelektronika\mikroC\Examples\EasyPic4\extra_examples\HID-
library\USBdsc.c"

unsigned char Read_buffer[64];
unsigned char Write_buffer[64];
unsigned char num;
//
// Timer interrupt service routine
//
void interrupt()
{
  HID_InterruptProc();                          // Keep alive
  TMR0L = 100;                                  // Re-load TMR0L
  INTCON.TMR0IF = 0;                            // Re-enable TMR0 interrupts
}


//
// Start of MAIN program
```

Figure 8.22: Microcontroller program with USB code

```
//
void main()
{

    ADCON1 = 0xFF;                                  // Set PORTB to digital I/O
    TRISB = 0;                                      // Set PORTB to outputs
    PORTB = 0;                                      // Clear all outputs
//
// Set interrupt registers to power-on defaults
// Disable all interrupts
//
    INTCON=0;
    INTCON2=0xF5;
    INTCON3=0xC0;
    RCON.IPEN=0;
    PIE1=0;
    PIE2=0;
    PIR1=0;
    PIR2=0;
//
// Configure TIMER 0 for 3.3ms interrupts. Set prescaler to 256
// and load TMR0L to 100 so that the time interval for timer
// interrupts at 48MHz is 256*(256-100)*0.083 = 3.3ms
//
// The timer is in 8-bit mode by default
//
  T0CON  = 0x47;                                    // Prescaler = 256
  TMR0L  = 100;                                     // Timer count is 256-156 = 100
  INTCON.TMR0IE = 1;                                // Enable T0IE
  T0CON.TMR0ON = 1;                                 // Turn Timer 0 ON
  INTCON = 0xE0;                                    // Enable interrupts

//
// Enable USB port
//
   Hid_Enable(&Read_buffer, &Write_buffer);
   Delay_ms(1000);
   Delay_ms(1000);
//
// Read from the USB port. Number of bytes read is in num
//

   for(;;)                                 // do forever
{
     num=0;
     while(num != 4)                       // Get 4 characters
     {num = Hid_Read();
     }
     if(Read_buffer[0] == 'P' && Read_buffer[1] == '=' && Read_buffer[3] == 'T')
     {
          PORTB = Read_buffer[2];
     }
 }
  Hid_Disable();

}
```

Figure 8.22: (Cont'd)

Figure 8.23 shows part of the PIC18F4550 clock circuit. The circuit consists of a 1:1 – 1:12 PLL prescaler and multiplexer, a 4:96MHz PLL, a 1:2 – 1:6 PLL postscaler, and a 1:1 – 1:4 oscillator postscaler. Assuming the crystal frequency is 8MHz and we want to operate the microcontroller with a 48MHz clock, and also remembering that a 48MHz clock is required for the USB module, we should make the following choices in the Edit Project option of the mikroC IDE:

- Set *_PLL_DIV2_1L* so the 8MHz clock is divided by 2 to produce 4MHZ at the output of the PLL prescaler multiplexer. The output of the 4:96MHZ PLL is now 96MHz. This is further divided by 2 to give 48MHz at the input of multiplexer USBDIV.

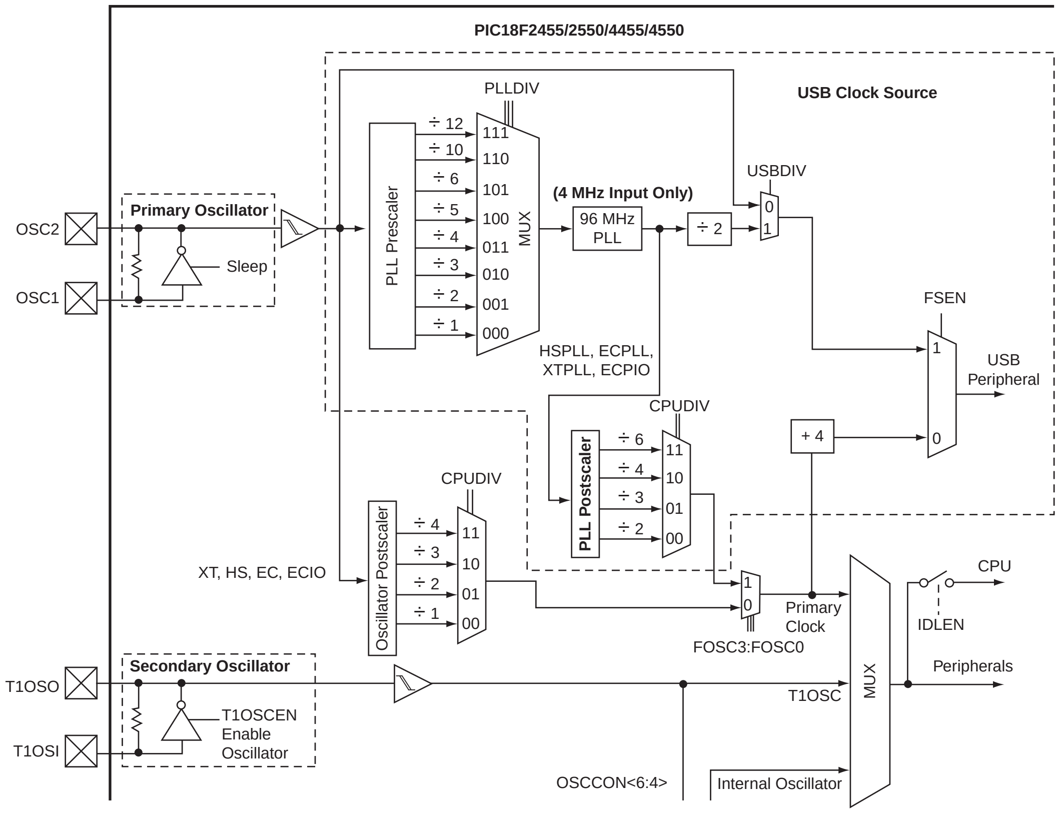


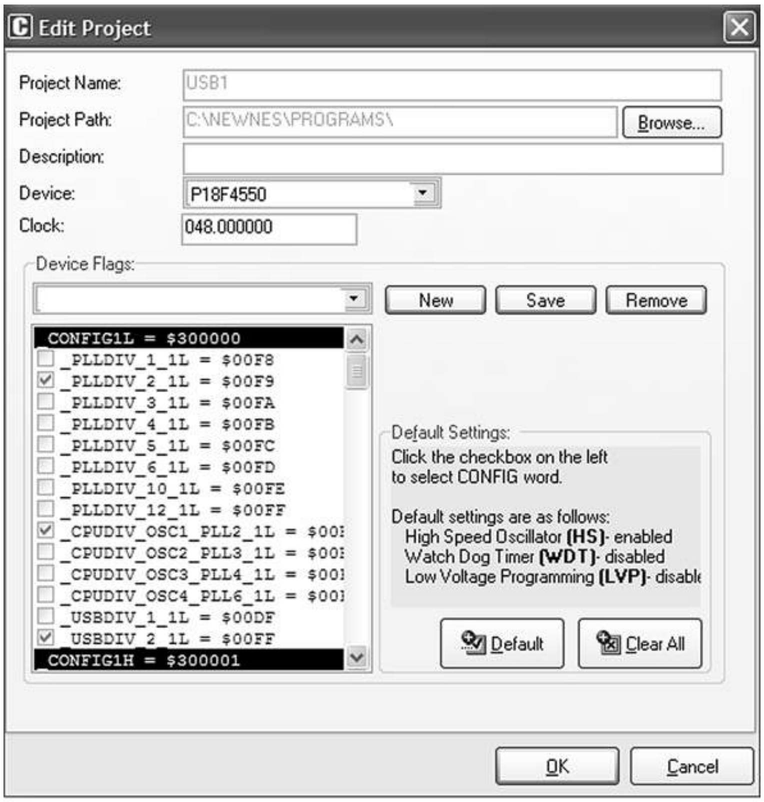
Figure 8.23: PIC18F4550 microcontroller clock

- Check *_USBDIV_2_1L* to provide a 48MHz clock to USB module and to select ÷2 for the PLL postscaler.
- Check *CPUDIV_OSC1_PLL2_1L* to select PLL as the clock source.
- Check *_FOSC_HSPLL_HS_1H* to select a 48MHz clock for the CPU.
- Set the CPU clock to 48MHz in mikroC IDE (using Edit Project).

The clock bits selected for the 48MHz USB operation with a 48MHz CPU clock are shown in Figure 8.24.

Setting other configuration bits in addition to the clock bits is recommended. The following list gives all the bits that should be set in the Edit Project option of the IDE (most of these settings are the power-on-reset values of the bits):

```
PLLDIV_2_1L
CPUDIV_OSC1_PLL2_1L
USBDIV_2_1L

FOSC_HSPLL_HS_1H
FCMEM_OFF_1H
IESO_OFF_1H

PWRT_ON_2L
BOR_ON_2L
BORV_43_2L
VREGEN_ON_2L

WDT_OFF_2H
WDTPS_256_2H

MCLRE_ON_3H
LPT1OSC_OFF_3H
PBADEN_OFF_3H
CCP2MX_ON_3H

STVREN_ON_4L
LVP_OFF_4L
ICPRT_OFF_4L
XINST_OFF_4L
DEBUG_OFF_4L
```



Edit Project

Project Name: USB1

Project Path: C:\NEWNES\PROGRAMS\

Description:

Device: P18F4550

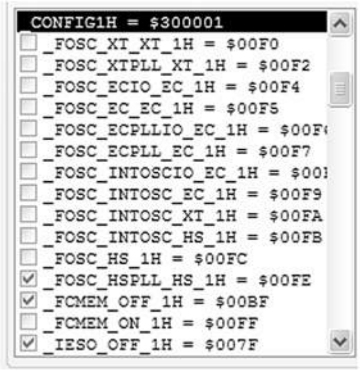
Clock: 048.000000

Device Flags:

```
CONFIG1L = $300000
[ ] _PLLDIV_1_1L = $00F8
[✓] _PLLDIV_2_1L = $00F9
[ ] _PLLDIV_3_1L = $00FA
[ ] _PLLDIV_4_1L = $00FB
[ ] _PLLDIV_5_1L = $00FC
[ ] _PLLDIV_6_1L = $00FD
[ ] _PLLDIV_10_1L = $00FE
[ ] _PLLDIV_12_1L = $00FF
[✓] _CPUDIV_OSC1_PLL2_1L = $00
[ ] _CPUDIV_OSC2_PLL3_1L = $00
[ ] _CPUDIV_OSC3_PLL4_1L = $00
[ ] _CPUDIV_OSC4_PLL6_1L = $00
[ ] _USBDIV_1_1L = $00DF
[✓] _USBDIV_2_1L = $00FF
CONFIG1H = $300001
```

Default Settings:
Click the checkbox on the left to select CONFIG word.

Default settings are as follows:
High Speed Oscillator **(HS)**- enabled
Watch Dog Timer **(WDT)**- disabled
Low Voltage Programming **(LVP)**- disabl



```
CONFIG1H = $300001
[ ] _FOSC_XT_XT_1H = $00F0
[ ] _FOSC_XTPLL_XT_1H = $00F2
[ ] _FOSC_ECIO_EC_1H = $00F4
[ ] _FOSC_EC_EC_1H = $00F5
[ ] _FOSC_ECPLLIO_EC_1H = $00F
[ ] _FOSC_ECPLL_EC_1H = $00F7
[ ] _FOSC_INTOSCIO_EC_1H = $00
[ ] _FOSC_INTOSC_EC_1H = $00F9
[ ] _FOSC_INTOSC_XT_1H = $00FA
[ ] _FOSC_INTOSC_HS_1H = $00FB
[ ] _FOSC_HS_1H = $00FC
[✓] _FOSC_HSPLL_HS_1H = $00FE
[✓] _FCMEM_OFF_1H = $00BF
[ ] _FCMEM_ON_1H = $00FF
[✓] _IESO_OFF_1H = $007F
```

Figure 8.24: Selecting clock bits for USB operation

Testing the Project

Testing the project is relatively easy. The steps are:

- Construct the hardware
- Load the program (Figure 8.22) into the PIC18F4550 microcontroller
- Copy or run the PC-based Visual Basic program

When the microcontroller is connected to one of the USB ports of the PC, a message should be visible at the bottom right-hand corner of the screen similar to the one in Figure 8.25. This message shows that the new USB HID device has been plugged in and is recognized by the PC.

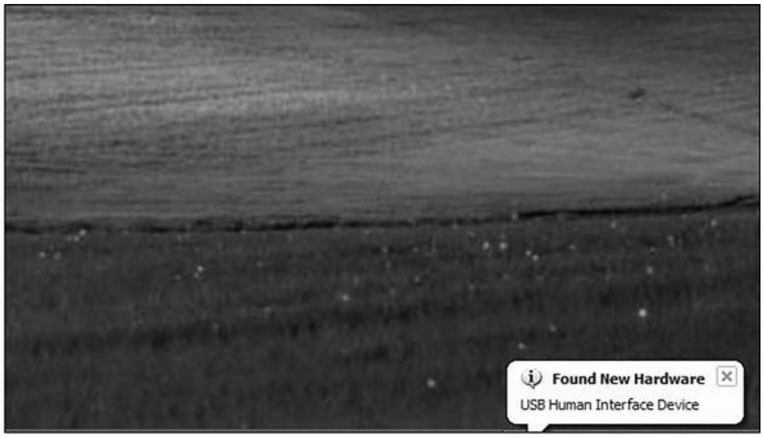

Found New Hardware
USB Human Interface Device

Figure 8.25: USB connection message

In addition, the device manager display should show an HID-compliant device and a USB human interface device as in Figure 8.26. The properties of these drivers can be displayed to make sure the VIP is 0 × 1234 and the PID is 1.

Enter data into the Visual Basic form and click the CLICK TO SEND button. The corresponding microcontroller LEDs should turn on. For example, entering 3 should turn on LEDs 0 and 1.

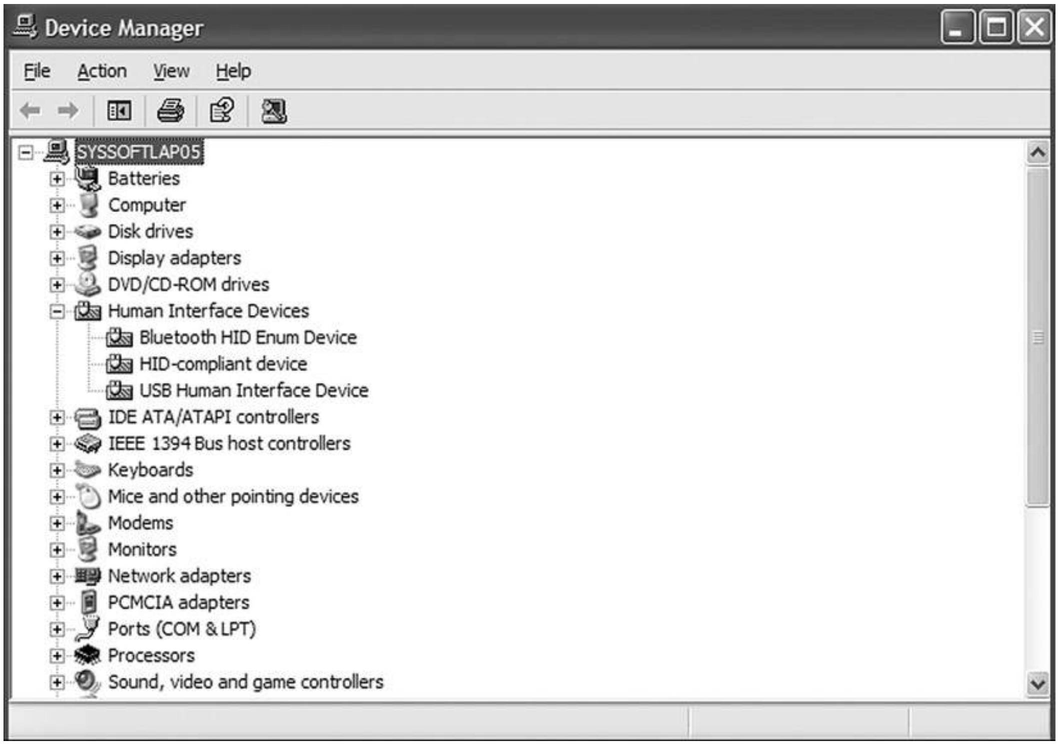


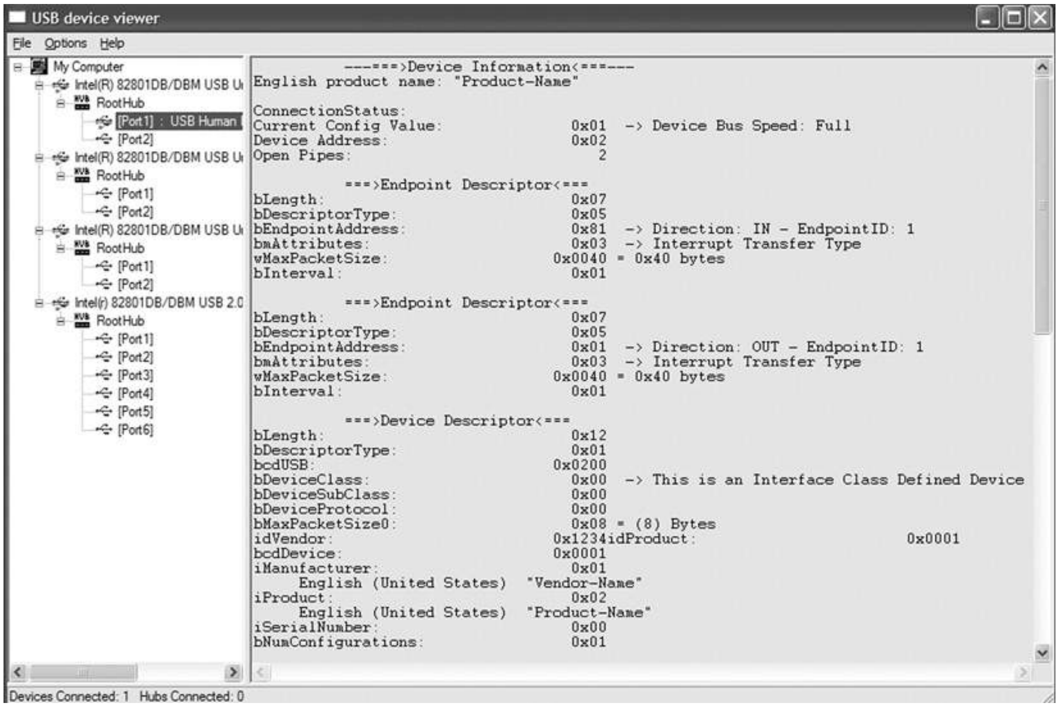
Figure 8.26: Device manager display showing the USB devices

Using a USB Protocol Analyzer

If for any reason the project is not working, a USB protocol analyzer can be used to check the data transactions on the USB bus. There are many USB protocol analyzers on the market. Some expensive professional ones are hardware-based and require the purchase of special hardware. Most low-cost USB protocol analyzers are software-based. Two such tools are described here briefly.

UVCView

UVCView is a free Microsoft product that runs on a PC and displays the descriptors of a USB device after it is plugged in. Figure 8.27 shows the UVCView display after the



```
          ---===>Device Information<===---
English product name: "Product-Name"

ConnectionStatus:
Current Config Value:              0x01  -> Device Bus Speed: Full
Device Address:                    0x02
Open Pipes:                           2

          ===>Endpoint Descriptor<===
bLength:                           0x07
bDescriptorType:                   0x05
bEndpointAddress:                  0x81  -> Direction: IN - EndpointID: 1
bmAttributes:                      0x03  -> Interrupt Transfer Type
wMaxPacketSize:                  0x0040 = 0x40 bytes
bInterval:                         0x01

          ===>Endpoint Descriptor<===
bLength:                           0x07
bDescriptorType:                   0x05
bEndpointAddress:                  0x01  -> Direction: OUT - EndpointID: 1
bmAttributes:                      0x03  -> Interrupt Transfer Type
wMaxPacketSize:                  0x0040 = 0x40 bytes
bInterval:                         0x01

          ===>Device Descriptor<===
bLength:                           0x12
bDescriptorType:                   0x01
bcdUSB:                          0x0200
bDeviceClass:                      0x00  -> This is an Interface Class Defined Device
bDeviceSubClass:                   0x00
bDeviceProtocol:                   0x00
bMaxPacketSize0:                   0x08 = (8) Bytes
idVendor:                        0x1234idProduct:                    0x0001
bcdDevice:                       0x0001
iManufacturer:                     0x01
     English (United States)  "Vendor-Name"
iProduct:                          0x02
     English (United States)  "Product-Name"
iSerialNumber:                     0x00
bNumConfigurations:                0x01
```

Figure 8.27: UVCView display of the project

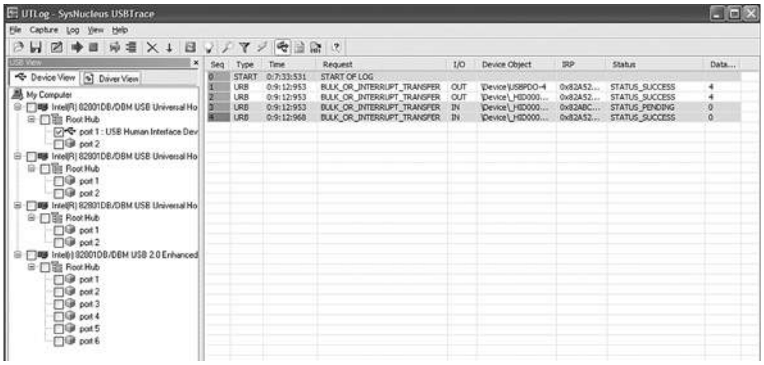
microcontroller is plugged into the PC. The left side of the display shows the USB ports available in the system. Clicking on a device in this part of the display shows descriptor details of the device in the middle of the screen. In Figure 8.27 the descriptors of our device are shown. The UVCView display is useful when various fields of the device descriptors must be checked.

USBTrace

USBTrace is a software USB protocol analyzer developed by SysNucleus (www.sysnucleus.com) and runs on a PC. The software monitors the USB ports of the PC it is running on and displays all the transactions on the bus. This software can be an invaluable tool when all the transactions on the line must be monitored and logged.

A limited-time demo version of USBTrace is available on the manufacturer's web site. An example using the program is given in this section to show the data sent from the PC to the microcontroller:

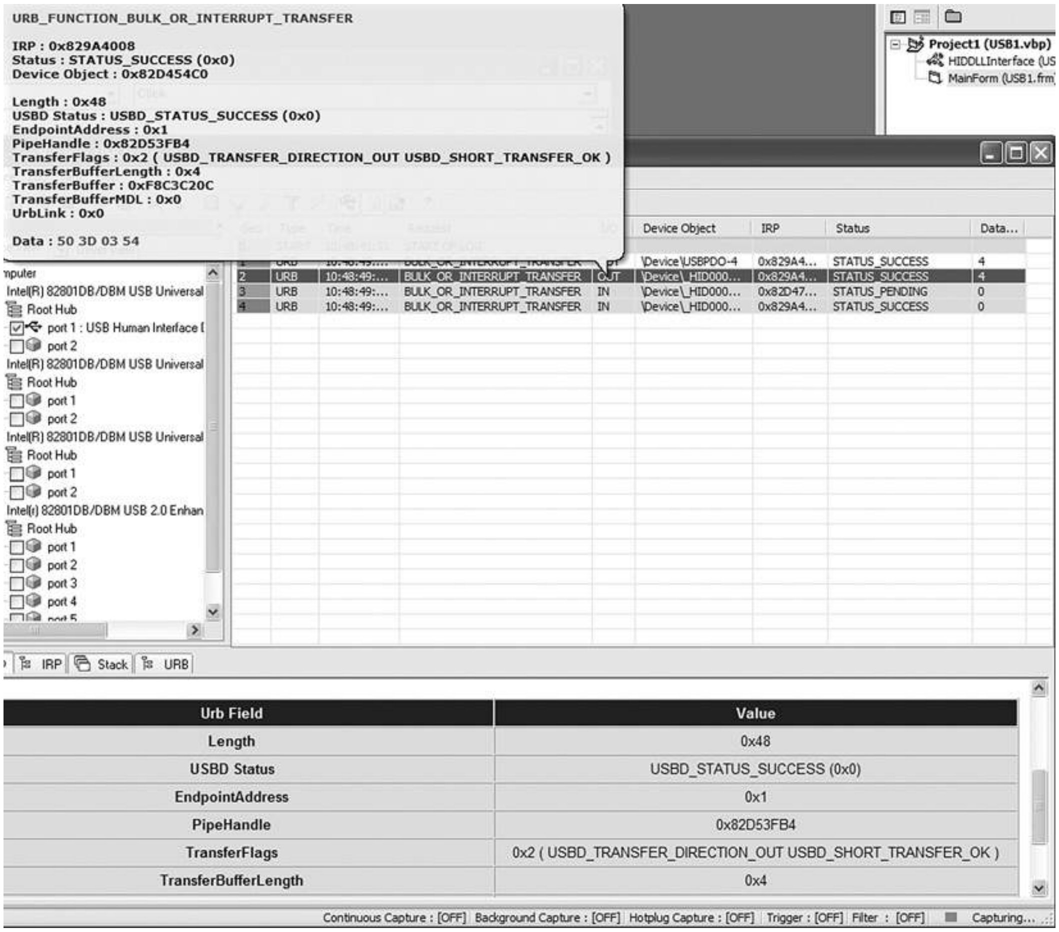
- Start the USBTrace program.
- Connect the microcontroller to the USB port of the PC.
- Select the device from the left side of the display by checking the appropriate box.
- Start the Visual Basic program.
- Start capturing data by clicking the green arrow at the top left of the USBTrace menu. You should see the START OF LOG message in the middle part of the screen
- Enter number 3 on the Visual Basic form to turn on LEDs 0 and 1 of PORTB, and click the CLICK TO SEND button.
- You should see data packets in the middle of the screen as shown in Figure 8.28.


UTLog - SysNucleus USBTrace
File Capture Log View Help
Device View
Driver View
My Computer
Intel(R) 82801DB/DBM USB Universal Ho
Root Hub
port 1 : USB Human Interface Dev
port 2
Intel(R) 82801DB/DBM USB 2.0 Enhanced

Seq	Type	Time	Request	I/O	Device Object	IRP	Status	Data...
0	START	0:7:33:531	START OF LOG					
1	URB	0:9:12:953	BULK_OR_INTERRUPT_TRANSFER	OUT	\Device\USBPDO-4	0x82A52...	STATUS_SUCCESS	4
2	URB	0:9:12:953	BULK_OR_INTERRUPT_TRANSFER	OUT	\Device_HID000...	0x82A52...	STATUS_SUCCESS	4
3	URB	0:9:12:953	BULK_OR_INTERRUPT_TRANSFER	IN	\Device_HID000...	0x82ABC...	STATUS_PENDING	0
4	URB	0:9:12:968	BULK_OR_INTERRUPT_TRANSFER	IN	\Device_HID000...	0x82A52...	STATUS_SUCCESS	0

Figure 8.28: Transactions on the bus when CLICK TO SEND is clicked

- Move the cursor over the first packet. This is the packet sent from the PC to the microcontroller (OUT packet). A pop-up window will appear, and information about this packet will be displayed, with the data sent appearing in hexadecimal


Project1 (USB1.vbp)
HIDDLLInterface (US
MainForm (USB1.frm)
mputer
Intel(R) 82801DB/DBM USB Universal
Root Hub
port 1 : USB Human Interface
port 2
Intel(R) 82801DB/DBM USB Universal
Root Hub
port 1
port 2
Intel(R) 82801DB/DBM USB Universal
Root Hub
port 1
port 2
Intel(r) 82801DB/DBM USB 2.0 Enhan
Root Hub
port 1
port 2
port 3
port 4
port 5
IRP
Stack
URB
Continuous Capture : [OFF] Background Capture : [OFF] Hotplug Capture : [OFF] Trigger : [OFF] Filter : [OFF] Capturing...

```
URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER

IRP : 0x829A4008
Status : STATUS_SUCCESS (0x0)
Device Object : 0x82D454C0

Length : 0x48
USBD Status : USBD_STATUS_SUCCESS (0x0)
EndpointAddress : 0x1
PipeHandle : 0x82D53FB4
TransferFlags : 0x2 ( USBD_TRANSFER_DIRECTION_OUT USBD_SHORT_TRANSFER_OK )
TransferBufferLength : 0x4
TransferBuffer : 0xF8C3C20C
TransferBufferMDL : 0x0
UrbLink : 0x0

Data : 50 3D 03 54
```

					Device Object	IRP	Status	Data...
1	URB	10:48:49:...	BULK_OR_INTERRUPT_TRANSFER	OUT	\Device\USBPDO-4	0x829A4...	STATUS_SUCCESS	4
2	URB	10:48:49:...	BULK_OR_INTERRUPT_TRANSFER	OUT	\Device_HID000...	0x829A4...	STATUS_SUCCESS	4
3	URB	10:48:49:...	BULK_OR_INTERRUPT_TRANSFER	IN	\Device_HID000...	0x82D47...	STATUS_PENDING	0
4	URB	10:48:49:...	BULK_OR_INTERRUPT_TRANSFER	IN	\Device_HID000...	0x829A4...	STATUS_SUCCESS	0

Urb Field	Value
Length	0x48
USBD Status	USBD_STATUS_SUCCESS (0x0)
EndpointAddress	0x1
PipeHandle	0x82D53FB4
TransferFlags	0x2 (USBD_TRANSFER_DIRECTION_OUT USBD_SHORT_TRANSFER_OK)
TransferBufferLength	0x4

Figure 8.29: Displaying contents of the packet

at the bottom of the display, as shown in Figure 8.29. Note that the data consists of the following 4 bytes:

```
50 3D 03 54
P = 3T
```

which correspond to the ASCII string P = 3T. This is the actual packet sent from the PC to the microcontroller.

USBTrace can also display the device descriptors in detail, as shown in the lower part of the screen in Figure 8.29.

Using the HID Terminal of mikroC

The mikroC IDE provides a USB terminal interface that can be used for sending and receiving data over the USB bus. This program can be used instead of the Visual Basic program to test the USB interface. The steps are as follows:

- In mikroC IDE, Select *Tools -> HID Terminal*
- Plug the microcontroller into the PC's USB port
- You should see the product ID under HID Devices:
    - To turn on LEDs 0,1,4, and 5, type P = 3T under Communication and click the SEND button as shown in Figure 8.30 (remember that the ASCII value of number 3 has the bit pattern "0011 0011")
    - LEDs 0,1,4, and 5 of the microcontroller should turn on

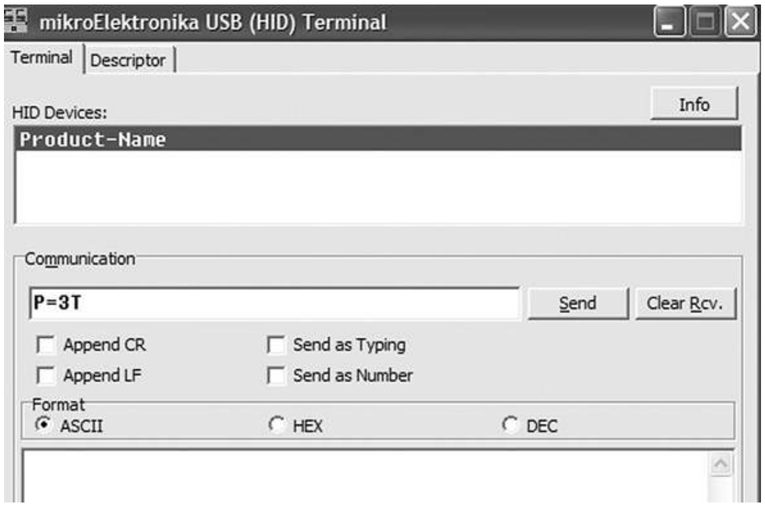


Figure 8.30: Using the HID terminal to send data to a USB device

PROJECT 8.2—USB-Based Microcontroller Input/Output

This project is very similar to Project 8.1, except that it includes two-way communication, while in Project 8.1 data to be output on PORTB was sent to the microcontroller. In addition, PORTB data is received from the microcontroller and displayed on the PC.

The PC sends two commands to the microcontroller:

- Command $P = nT$ requests the microcontroller to send data byte n to PORTB.
- Command $P = ??$ requests the microcontroller to read its PORTB data and send it as a byte to the PC. The PC then displays this data on the screen. The microcontroller sends its data in the familiar format P = nT.

The hardware of this project is the same as the hardware for the previous project, shown in Figure 8.11, where eight LEDs are connected to PORTB of a PIC18F4550 microcontroller which is operated from a 8MHz crystal.

A single form is used in this project, and Figure 8.31 shows the format of this form. The upper part of the form is the same as in Project 8.1, i.e., sending data to PORTB of the microcontroller. A text box and a command button named CLICK TO RECEIVE are also placed on the form. When the button is pressed, the PC sends command $P = ??$ to the microcontroller. The microcontroller reads its PORTB data and sends it in the format P = nT to the PC where it is displayed in the text box.

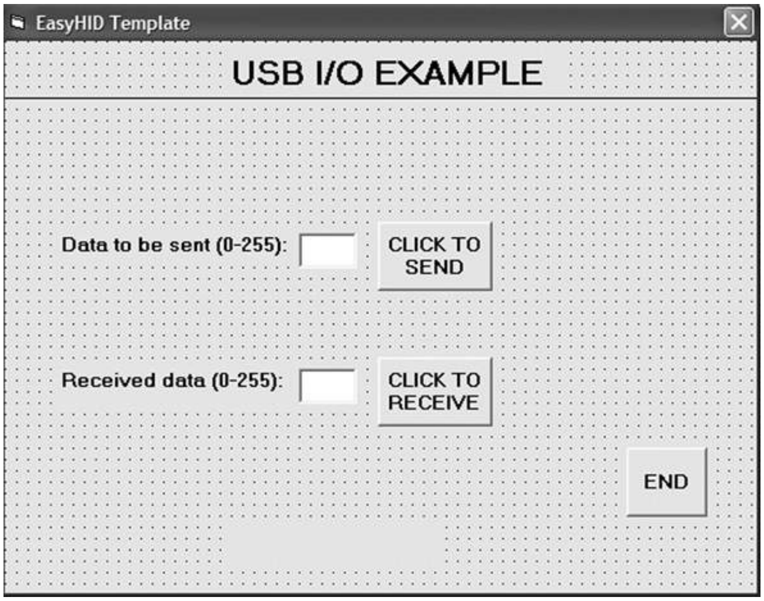


Figure 8.31: Visual Basic form of the project

Figure 8.32 shows the mikroC program of the project. The program is named USB2.C and is very similar to the one for the previous project. But here, in addition, when the command *P = ??* is received from the PC, the microcontroller reads PORTB data and sends it to the PC in the format using the mikroC function *Hid_Write*.

The program checks the format of the received command. For P = ?? type commands, PORTB is configured as inputs, PORTB data is read into *Write_buffer[2]*, and *Write_buffer* is sent to the PC, where *Write_buffer[0]* = "P," *Write_buffer[1]* = "=", and *Write_buffer[3]* = "T" as follows:

```
if(Read_buffer[0] == 'P' && Read_buffer[1] == '=' &&
Read_buffer[2] == '?' && Read_Buffer[3] == '?')
{
TRISB = 0×FF;
Write_buffer[0] = 'P'; Write_buffer[1] = '='; Write_buffer[2] =
  PORTB; Write_buffer[3] = 'T';
Hid_Write(&Write_buffer,4);
}
```

For P = nT type commands, PORTB is configured as outputs and *Read_buffer[2]* is sent to PORTB as follows:

```
if(Read_buffer[0] == 'P' && Read_buffer[1] == '=' &&
Read_buffer[3] == 'T')
{
TRISB = 0;
PORTB = Read_buffer[2];
}
```

The microcontroller clock should be set as in Project 8.1 (i.e., both the CPU and the USB module should have 48MHz clocks). The other configurations bits should also be set as described in the previous problem.

Testing the Project

The project can be tested using one of the methods described in the previous project. If you are using the Visual Basic program, send data to the microcontroller and make sure the correct LEDs are turned on. Then connect some of the PORTB pins to logic 0 and click the CLICK TO RECEIVE button. The microcontroller will read its PORTB data and send it to the PC, where it will be displayed on the PC screen.

```
/****************************************************************************
            USB BASED MICROCONTROLLER INPUT/OUTPUT PORT
            ============================================

In this project a PIC18F4550 type microcontroller is connected
to a PC through the USB link.

A Visual Basic program runs on the PC where the user enters the
bits to be set or cleared on PORTB of the microcontroller. The
PC sends a command to the microcontroller requesting it to set
or reset the required bits of the microcontroller PORTB. In addition,
the PORTB data can be requested from the microcontroller and displayed
on the PC.

The microcontroller is operated from a 8MHz crystal, but the CPU
clock frequency is increased to 48MHz. Also, the USB module operates
with 48MHz.

The commands are:

From PC to microcontroller: P=nT (Send data byte n to PORTB)
                            P=?? (Give me PORTB data)

From microcontroller to PC: P=nT (Here is my PORTB data)

Author:     Dogan Ibrahim
Date:       September 2007
File:       USB2.C
*****************************************************************************/

#include "C:\Program
Files\Mikroelektronika\mikroC\Examples\EasyPic4\extra_examples\HID-
library\USBdsc.c"

unsigned char Read_buffer[64];
unsigned char Write_buffer[64];
unsigned char num,i;
//
// Timer interrupt service routine
//
void interrupt()
{
  HID_InterruptProc();                          // Keep alive
  TMR0L = 100;                                  // Reload TMR0L
  INTCON.TMR0IF = 0;                            // Re-enable TMR0 interrupts
}


//
// Start of MAIN program
```

Figure 8.32: mikroC program listing of the project

(Continued)

```
//
void main()
{

     ADCON1 = 0xFF;                    // Set PORTB to digital I/O
     TRISB = 0;                        // Set PORTB to outputs
     PORTB = 0;                        // PORTB all 0s to start with

//
// Set interrupt registers to power-on defaults
// Disable all interrupts
//
     INTCON=0;
     INTCON2=0xF5;
     INTCON3=0xC0;
     RCON.IPEN=0;
     PIE1=0;
     PIE2=0;
     PIR1=0;
     PIR2=0;
//
// Configure TIMER 0 for 20ms interrupts. Set prescaler to 256
// and load TMR0L to 156 so that the time interval for timer
// interrupts at 8MHz is 256*156*0.5 = 20ms
//
// The timer is in 8-bit mode by default
//
  T0CON  = 0x47;                       // Prescaler = 256
  TMR0L  = 100;                        // Timer count is 256-156 = 100
  INTCON.TMR0IE = 1;                   // Enable T0IE
  T0CON.TMR0ON = 1;                    // Turn Timer 0 ON
  INTCON = 0xE0;                       // Enable interrupts

//
// Enable USB port
//
   Hid_Enable(&Read_buffer, &Write_buffer);
   Delay_ms(1000);
   Delay_ms(1000);
//
// Read from the USB port. Number of bytes read is in num
//

   for(;;)                             // do forever
{
      num=0;
      while(num != 4)
      {num = Hid_Read();
      }
```

Figure 8.32: (Cont'd)

```
      if(Read_buffer[0] == 'P' && Read_buffer[1] == '=' &&
        Read_buffer[2] == '?' && Read_Buffer[3] == '?')
      {
        TRISB = 0xFF;
        Write_buffer[0] = 'P';    Write_buffer[1] = '=';
        Write_buffer[2] = PORTB;  Write_buffer[3] = 'T';
        Hid_Write(&Write_buffer,4);
      }
      else
      {
        if(Read_buffer[0] == 'P' && Read_buffer[1] == '=' &&
          Read_buffer[3] == 'T')
        {
          TRISB = 0;
          PORTB = Read_buffer[2];
        }
      }
  }
    Hid_Disable();

}
```

Figure 8.32: (Cont'd)

The project can also be tested using the HID terminal of mikroC IDE. The steps are:

- Start the HID terminal.
- Send a command to the microcontroller to turn on the LEDs (e.g., $P = 1T$) and make sure the correct LEDs are turned on (in this case, LEDs 0, 4, and 5 should turn on, corresponding to the data pattern "0011 0001").
- Connect bits 2 and 3 of PORTB to logic 1 and the other six bits to ground.
- Send command $P = ??$ to the microcontroller.
- The PC will display the number 12, corresponding to bit pattern "0000 1100".

The Visual Basic program listing of the project is given in Figure 8.33. Only the main program is given here, as the library declarations are the same as in Figure 8.19. The program jumps to subroutine *OnRead* when data arrives at the USB bus. The format of this data is checked to be in the format P = nT, and if the format is correct, the received data byte is displayed in the text box.

An installable version of the Visual Basic PC program is available in folder USB2 on the CDROM included with this book.

CAN Bus: Introduction
Frames
Bit stuffing
Types of errors
Nominal Bit Timing
PIC microcontroller CAN Interface
A simple application with CAN

Advanced PIC18 Projects—CAN Bus Projects

The Controller Area Network (CAN) is a serial bus communications protocol developed by Bosch (an electrical equipment manufacturer in Germany) in the early 1980s. Thereafter, CAN was standardized as ISO-11898 and ISO-11519, establishing itself as the standard protocol for in-vehicle networking in the auto industry. In the early days of the automotive industry, localized stand-alone controllers had been used to manage various actuators and electromechanical subsystems. By networking the electronics in vehicles with CAN, however, they could be controlled from a central point, the engine control unit (ECU), thus increasing functionality, adding modularity, and making diagnostic processes more efficient.

Early CAN development was mainly supported by the vehicle industry, as it was used in passenger cars, boats, trucks, and other types of vehicles. Today the CAN protocol is used in many other fields in applications that call for networked embedded control, including industrial automation, medical applications, building automation, weaving machines, and production machinery. CAN offers an efficient communication protocol between sensors, actuators, controllers, and other nodes in real-time applications, and is known for its simplicity, reliability, and high performance.

The CAN protocol is based on a bus topology, and only two wires are needed for communication over a CAN bus. The bus has a multimaster structure where each device on the bus can send or receive data. Only one device can send data at any time while all the others listen. If two or more devices attempt to send data at the same time, the one with the highest priority is allowed to send its data while the others return to receive mode.

As shown in Figure 9.1, in a typical vehicle application there is usually more than one CAN bus, and they operate at different speeds. Slower devices, such as door control, climate control, and driver information modules, can be connected to a slow speed bus. Devices that require faster response, such as the ABS antilock braking system, the transmission control module, and the electronic throttle module, are connected to a faster CAN bus.

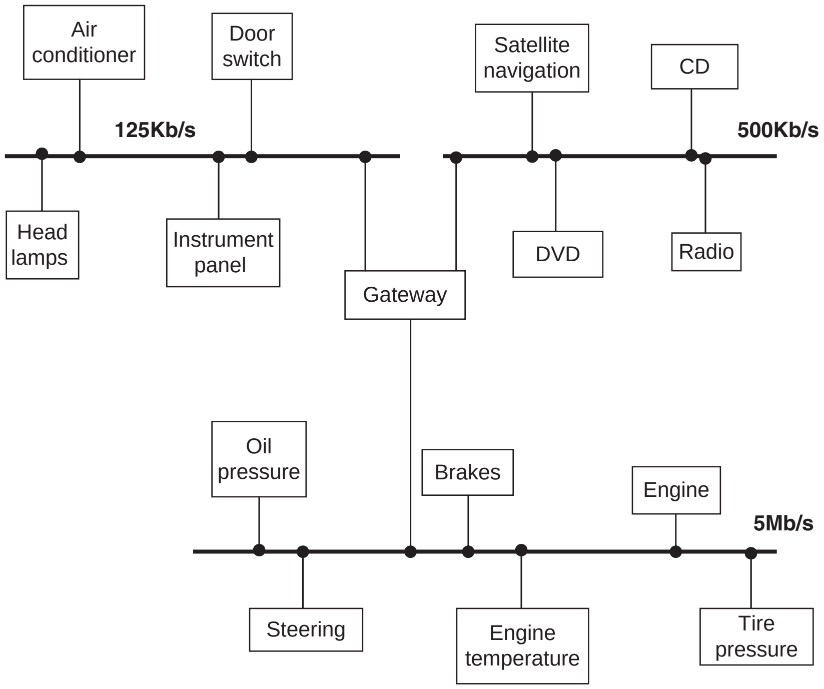


Figure 9.1: Typical CAN bus application in a vehicle

The automotive industry's use of CAN has caused mass production of CAN controllers. Current estimate is that 400 million CAN modules are sold every year, and CAN controllers are integrated on many microcontrollers, including PIC microcontrollers, and are available at low cost.

Figure 9.2 shows a CAN bus with three nodes. The CAN protocol is based on CSMA/CD+AMP (Carrier-Sense Multiple Access/Collision Detection with Arbitration on Message Priority) protocol, which is similar to the protocol used in Ethernet LAN. When Ethernet detects a collision, the sending nodes simply stop transmitting and wait

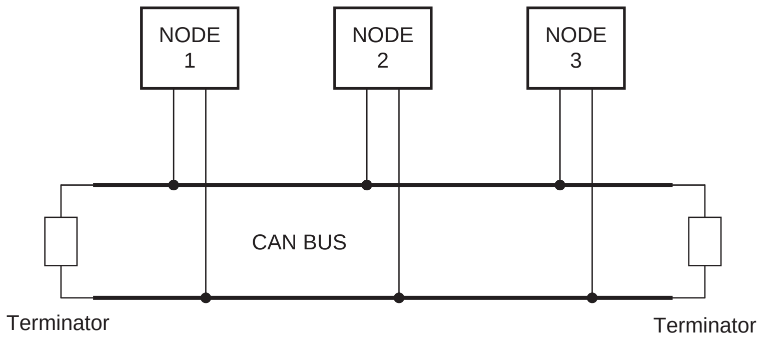


Figure 9.2: Example CAN bus

a random amount of time before trying to send again. CAN protocol, however, solves the collision problem using the principle of arbitration, where only the higheest priority node is given the right to send its data.

There are basically two types of CAN protocols: 2.0A and 2.0B. CAN 2.0A is the earlier standard with 11 bits of identifier, while CAN 2.0B is the new extended standard with 29 bits of identifier. 2.0B controllers are completely backward-compatible with 2.0A controllers and can receive and transmit messages in either format.

There are two types of 2.0A controllers. The first is capable of sending and receiving 2.0A messages only, and reception of a 2.0B message will flag an error. The second type of 2.0A controller (known as 2.0B passive) sends and receives 2.0A messages but will also acknowledge receipt of 2.0B messages and then ignore them.

Some of the CAN protocol features are:

- CAN bus is multimaster. When the bus is free, any device attached to the bus can start sending a message.
- CAN bus protocol is flexible. The devices connected to the bus have no addresses, which means messages are not transmitted from one node to another based on addresses. Instead, all nodes in the system receive every message transmitted on the bus, and it is up to each node to decide whether the received message should be kept or discarded. A single message can be destined for a particular node or for many nodes, depending on how the system is designed. Another advantage of having no addresses is that when a device is added to or

removed from the bus, no configuration data needs to be changed (i.e., the bus is "hot pluggable").

- CAN bus offers remote transmit request (RTR), which means that one node on the bus is able to request information from the other nodes. Thus instead of waiting for a node to continuously send information, a request for information can be sent to the node. For example, in a vehicle, where the engine temperature is an important parameter, the system can be designed so the temperature is sent periodically over the bus. However, a more elegant solution is to request the temperature as needed, since it minimizes the bus traffic while maintaining the network's integrity.
- CAN bus communication speed is not fixed. Any communication speed can be set for the devices attached to a bus.
- All devices on the bus can detect an error. The device that has detected an error immediately notifies all other devices.
- Multiple devices can be connected to the bus at the same time, and there are no logical limits to the number of devices that can be connected. In practice, the number of units that can be attached to a bus is limited by the bus's delay time and electrical load.

The data on CAN bus is differential and can be in two states: dominant and recessive. Figure 9.3 shows the state of voltages on the bus. The bus defines a logic bit 0 as a dominant bit and a logic bit 1 as a recessive bit. When there is arbitration on the bus, a

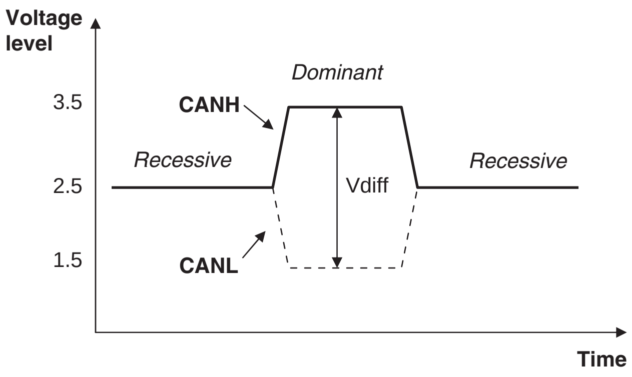


Figure 9.3: CAN logic states

dominant bit state always wins out over a recessive bit state. In the recessive state, the differential voltage CANH and CANL is less than the minimum threshold (i.e., less than 0.5V receiver input and less than 1.5V transmitter output). In the dominant state, the differential voltage CANH and CANL is greater than the minimum threshold.

The ISO-11898 CAN bus specifies that a device on that bus must be able to drive a forty-meter cable at 1Mb/s. A much longer bus length can usually be achieved by lowering the bus speed. Figure 9.4 shows the variation of bus length with the communication speed. For example, with a bus length of one thousand meters we can have a maximum speed of 40Kb/s.

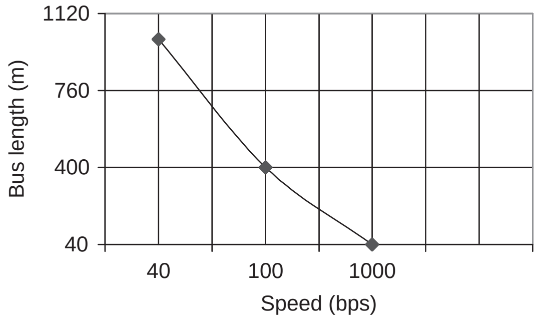


Figure 9.4: CAN bus speed and bus length

A CAN bus is terminated to minimize signal reflections on the bus. The ISO-11898 requires that the bus has a characteristic impedance of 120 ohms. The bus can be terminated by one of the following methods:

- Standard termination
- Split termination
- Biased split termination

In *standard termination*, the most common termination method, a 120-ohm resistor is used at each end of the bus, as shown in Figure 9.5(a). In *split termination*, the ends of the bus are split and a single 60-ohm resistor is used as shown in Figure 9.5(b). Split termination allows for reduced emission, and this method is gaining popularity. *Biased split termination* is similar to split termination except that a voltage divider

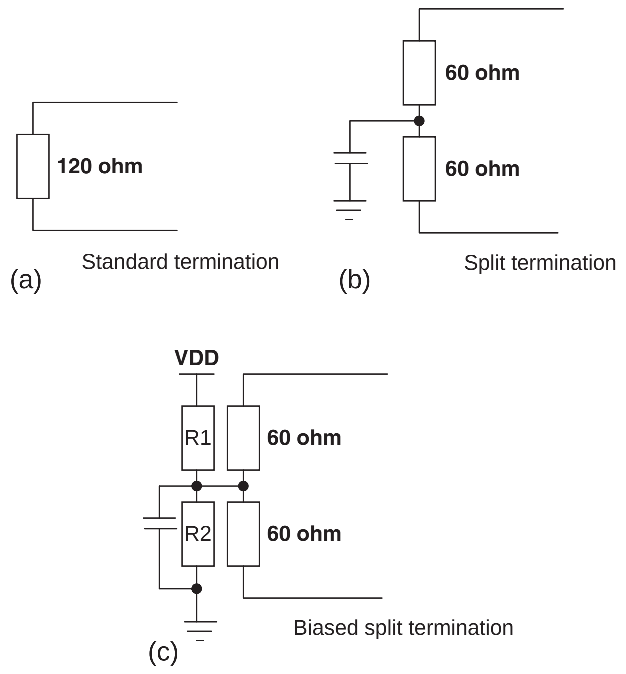


Figure 9.5: Bus termination methods

circuit and a capacitor are used at either end of the bus. This method increases the EMC performance of the bus (Figure 9.5(c)).

Many network protocols are described using the seven-layer Open Systems Interconnection (OSI) model. The CAN protocol includes the data link layer, and the physical layer of the OSI reference model (see Figure 9.6). The data link layer (DLL) consists of the Logical Link Control (LLC) and Medium Access Control (MAC). LLC manages the overload notification, acceptance filtering, and recovery management. MAC manages the data encapsulation, frame coding, error detection, and serialization/deserialization of the data. The physical layer consists of the physical signaling layer (PSL), physical medium attachment (PMA), and the medium dependent interface (MDI). PSL manages the bit encoding/decoding and bit timing. PMA manages the driver/receiver characteristics, and MDI is the connections and wires.

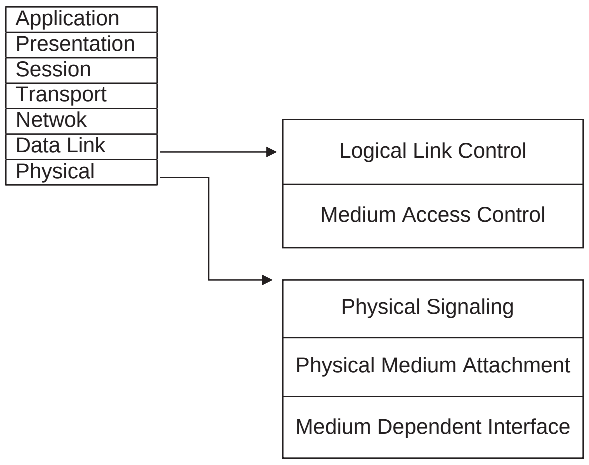


Figure 9.6: CAN and the OSI model

There are basically four message frames in CAN: data, remote, error, and overload. The data and remote frames need to be set by the user. The other two are set by the CAN hardware.

9.1 Data Frame

The data frame is in two formats: standard (having an 11-bit ID) and extended (having a 29-bit ID). The data frame is used by the transmitting device to send data to the receiving device, and the data frame is the most important frame handled by the user. Figure 9.7 shows the data frame's structure. A standard data frame starts with the start of frame (SOF) bit, which is followed by an 11-bit identifier and the remote transmission request (RTR) bit. The identifier and the RTR form the 12-bit arbitration field. The control field is 6 bits wide and indicates how many bytes of data are in the data field. The data field can be 0 to 8 bytes. The data field is followed by the

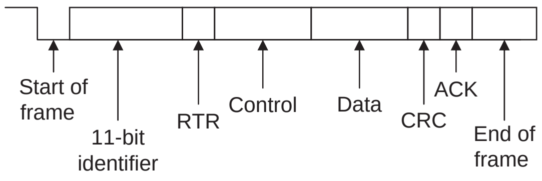


Figure 9.7: Standard data frame

CRC field, which checks whether or not the received bit sequence is corrupted. The ACK field is 2 bits and is used by the transmitter to receive acknowledgment of a valid frame from any receiver. The end of the message is indicated by a 7-bit end of frame (EOF) field. In an extended data frame, the arbitration field is 32 bits wide (29-bit identifier +1-bit IDE to define the message as an extended data frame +1-bit SRR which is unused +1-bit RTR) (see Figure 9.8).

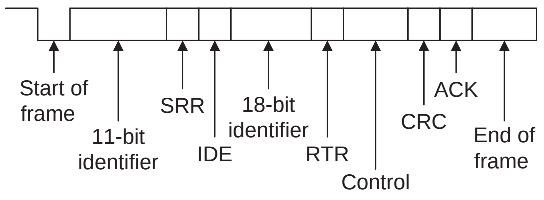


Figure 9.8: Extended data frame

The data frame consists of the following fields:

9.1.1 Start of Frame (SOF)

The start of frame field indicates the beginning of a data frame and is common to both standard and extended formats.

9.1.2 Arbitration Field

Arbitration is used to resolve bus conflicts that occur when several devices at once start sending messages on the bus. The arbitration field indicates the priority of a frame, and it is different in the standard and extended formats. In the standard format there are 11 bits, and up to 2032 IDs can be set. The extended format ID consists of 11 base IDs plus 18 extended IDs. Up to 2032×2^{18} discrete IDs can be set.

During the arbitration phase, each transmitting device transmits its identifier and compares it with the level on the bus. If the levels are equal, the device continues to transmit. If the device detects a dominant level on the bus while it is trying to transmit a recessive level, it quits transmitting and becomes a receiving device. After arbitration only one transmitter is left on the bus, and this transmitter continues to send its control field, data field, and other data.

The process of arbitration is illustrated in Figure 9.9 by an example consisting of three nodes having identifiers:

```
Node 1: 11100110011  Node 2: 11100111111  Node 3: 11100110001
```

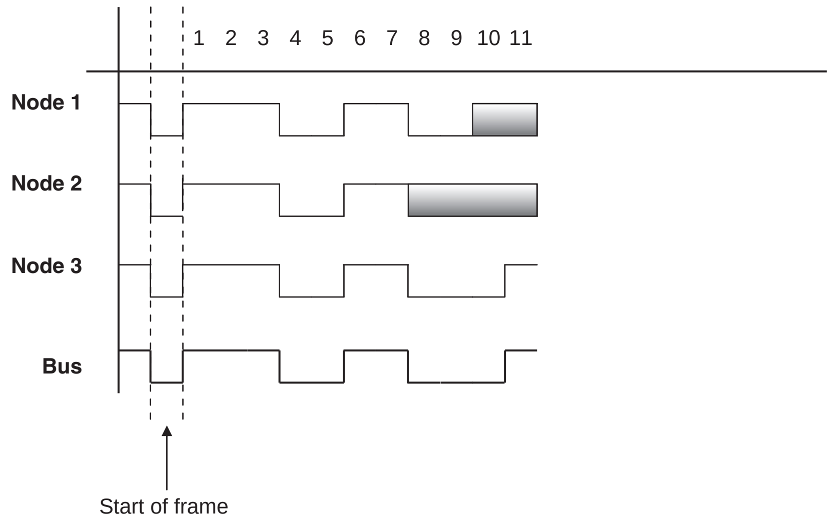


Figure 9.9: Example CAN bus arbitration

Assuming the recessive level corresponds to 1 and the dominant level to 0, the arbitration is performed as follows:

- All the nodes start transmitting simultaneously, first sending SOF bits.
- Then they send their identifier bits. The 8th bit of Node 2 is in the recessive state, while the corresponding bits of Nodes 1 and 3 are in the dominant state. Therefore Node 2 stops transmitting and returns to receive mode. The receiving phase is indicated by a gray field.
- The 10th bit of Node 1 is in the recessive state, while the same bit of Node 3 is in dominant state. Thus Node 1 stops transmitting and returns to receive mode.
- The bus is now left to Node 3, which can send its control and data fields freely.

Notably, the devices on the bus have no addresses. Instead, all the devices pick up all the data on the bus, and every node must filter out the messages it does not want.

9.1.3 Control Field

The control field is 6 bits wide, consisting of 2 reserved bits and 4 data length code (DLC) bits, and indicates the number of data bytes in the message being transmitted. This field is coded as shown in Table 9.1, where up to 8 transmit bytes can be coded with 6 bits.

Table 9.1: Coding the control field

No. of data bytes	DLC3	DLC2	DLC1	DLC0
0	D	D	D	D
1	D	D	D	R
2	D	D	R	D
3	D	D	R	R
4	D	R	D	D
5	D	R	D	R
6	D	R	R	D
7	D	R	R	R
8	R	D or R	D or R	D or R

D: Dominant level, R: Recessive level.

9.1.4 Data Field

The data field carries the actual content of the message. The data size can vary from 0 to 8 bytes. The data is transmitted with the MSB first.

9.1.5 CRC Field

The CRC field, consisting of a 15-bit CRC sequence and a 1-bit CRC delimiter, is used to check the frame for a transmission error. The CRC calculation includes the start of frame, arbitration field, control field, and data field. The calculated CRC and the received CRC sequence are compared, and if they do not match, an error is assumed.

9.1.6 ACK Field

The ACK field indicates that the frame has been received normally. This field consists of 2 bits, one for ACK slot and one for ACK delimiter.

9.2 Remote Frame

The remote frame is used by the receiving unit to request transmission of a message from the transmitting unit. It consists of six fields (see Figure 9.10): start of frame, arbitration field, control field, CRC field, ACK field, and end of frame field. A remote frame is the same as a data frame except that it lacks a data field.

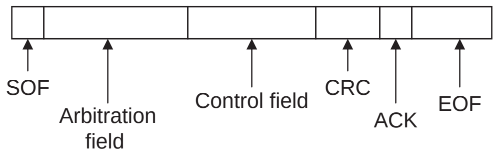


Figure 9.10: Remote frame

9.3 Error Frame

Error frames are generated and transmitted by the CAN hardware and are used to indicate when an error has occurred during transmission. An error frame consists of an error flag and an error delimiter. There are two types of error flags: active, which consists of 6 dominant bits, and passive, which consists of 6 recessive bits. The error delimiter consists of 8 recessive bits.

9.4 Overload Frame

The overload frame is used by the receiving unit to indicate that it is not yet ready to receive frames. This frame consists of an overload flag and an overload delimiter. The overload flag consists of 6 dominant bits and has the same structure as the active error flag of the error frame. The overload delimiter consists of 8 recessive bits and has the same structure as the error delimiter of the error frame.

9.5 Bit Stuffing

The CAN bus makes use of bit stuffing, a technique to periodically synchronize transmit-receive operations to prevent timing errors between receive nodes. After 5 consecutive bits with the same level, one bit of inverted data is added to the sequence. If, during sending of a data frame or remote frame, the same level occurs in 5 consecutive bits anywhere from the start of frame to the CRC sequence, an inverted bit is inserted in the next (i.e., the sixth) bit. If, during receiving of a data frame or remote frame, the same level occurs in 5 consecutive bits anywhere from the start of frame to CRC sequence, the next (sixth) bit is deleted from the received frame. If the deleted sixth bit is at the same level as the fifth bit, an error (stuffing error) is detected.

9.6 Types of Errors

The CAN bus identifies five types of errors:

- Bit error
- CRC error
- Form error
- ACK error
- Stuffing error

Bit errors are detected when the output level and the data level on the bus do not match. Both transmit and receive units can detect bit errors. *CRC errors* are detected only by receiving units. CRC errors are detected if the calculated CRC from the received message and the received CRC do not match. *Form errors* are detected by the transmitting or receiving units when an illegal frame format is detected. *ACK errors* are detected only by the transmitting units if the ACK field is found recessive. *Stuffing errors* are detected when the same level of data is detected for 6 consecutive bits in any field that should have been bit-stuffed. This error can be detected by both the transmitting and receiving units.

9.7 Nominal Bit Timing

The CAN bus nominal bit rate is defined as the number of bits transmitted every second without resynchronization. The inverse of the nominal bit rate is the nominal bit time. All devices on the CAN bus must use the same bit rate, even though each device can have its own different clock frequency. One message bit consists of four nonoverlapping time segments:

- Synchronization segment (Sync_Seg)
- Propagation time segment (Prop_Seg)
- Phase buffer segment 1 (Phase_Seg1)
- Phase buffer segment 2 (Phase_Seg2)

The *Sync_Seg* segment is used to synchronize various nodes on the bus, and an edge is expected to lie within this segment. The *Prop_Seg* segment compensates for physical delay times within the network. The *Phase_Seg1* and *Phase_Seg2* segments compensate for edge phase errors. These segments can be lengthened or shortened by synchronization. The sample point is the point in time where the actual bit value is located and occurs at the end of *Phase_Seg1*. A CAN controller can be configured to sample three times and use a majority function to determine the actual bit value.

Each segment is divided into units known as time quantum, or T_Q. A desired bit timing can be set by adjusting the number of T_Q's that comprise one message bit and the number of T_Q's that comprise each segment in it. The T_Q is a fixed unit derived from the oscillator period, and the time quantum of each segment can vary from 1 to 8. The lengths of the various time segments are:

- Sync_Seg is 1 time quantum long
- Prop_Seg is programmable as 1 to 8 time quanta long
- Phase_Seg1 is programmable as 1 to 8 time quanta long
- Phase_Seg2 is programmable as 2 to 8 time quanta long

By setting the bit timing, a sampling point can be set so multiple units on the bus can sample messages with the same timing.

The nominal bit time is programmable from a minimum of 8 time quanta to a maximum of 25 time quanta. By definition, the minimum nominal bit time is 1μs, corresponding to a maximum 1Mb/s rate. The nominal bit time (T_{BIT}) is given by:

$$T_{BIT} = T_Q * (\text{Sync\_Seg} + \text{Prop\_Seg} + \text{Phase\_Seg1} + \text{Phase\_Seg2}) \qquad (9.1)$$

and the nominal bit rate (NMR) is

$$NBR = 1/T_{BIT} \tag{9.2}$$

The time quantum is derived from the oscillator frequency and the programmable baud rate prescaler, with integer values from 1 to 64. The time quantum can be expressed as:

$$T_Q = 2 * (BRP + 1)/F_{OSC} \tag{9.3}$$

where T_Q is in µs, F_{OSC} is in MHz, and BRP is the baud rate prescaler (0 to 63).

Equation (9.2) can be written as

$$T_Q = 2 * (BRP + 1) * T_{OSC} \tag{9.4}$$

where T_{OSC} is in µs.

An example of the calculation of a nominal bit rate follows.

Example 9.1

Assuming a clock frequency of 20MHz, a baud rate prescaler value of 1, and a nominal bit time of $T_{BIT} = 8 * T_Q$, determine the nominal bit rate.

Solution 9.1

Using equation (9.3),

$$T_Q = 2 * (1 + 1)/20 = 0.2\mu s$$

also

$$T_{BIT} = 8 * T_Q = 8 * 0.2 = 1.6\mu s$$

From Equation (9.2),

$$NBR = 1/T_{BIT} = 1/1.6\mu s = 625,000 \text{bites/s or } 625\text{Kb/s}$$

In order to compensate for phase shifts between the oscillator frequencies of nodes on a bus, each CAN controller must synchronize to the relevant signal edge of the received signal. Two types of synchronization are defined: hard synchronization and resynchronization. Hard synchronization is used only at the beginning of a message frame, when each CAN node aligns the *Sync_Seg* of its current bit time to the recessive or dominant edge of the transmitted start of frame. According to the rules of synchronization, if a hard synchronization occurs, there will not be a resynchronization within that bit time.

With resynchronization, *Phase_Seg1* may be lengthened or *Phase_Seg2* may be shortened. The amount of change in the phase buffer segments has an upper bound given by the synchronization jump width (SJW). The SJW is programmable between 1 and 4, and its value is added to *Phase_Seg1* or subtracted from *Phase_Seg2*.

9.8 PIC Microcontroller CAN Interface

In general, any type of PIC microcontroller can be used in CAN bus–based projects, but some PIC microcontrollers (e.g., PIC18F258) have built-in CAN modules, which can simplify the design of CAN bus–based systems. Microcontrollers with no built-in CAN modules can also be used in CAN bus applications, but additional hardware and software are required, making the design costly and also more complex.

Figure 9.11 shows the block diagram of a PIC microcontroller–based CAN bus application, using a PIC16 or PIC12-type microcontroller (e.g., PIC16F84) with no

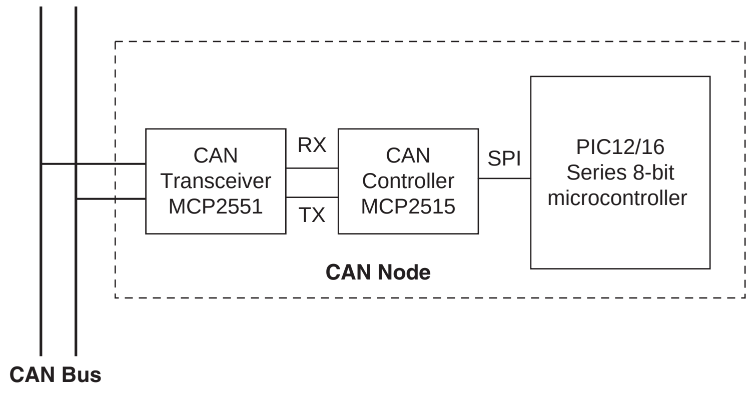


Figure 9.11: CAN node with any PIC microcontroller

built-in CAN module. The microcontroller is connected to the CAN bus using an external MCP2515 CAN controller chip and an MCP2551 CAN bus transceiver chip. This configuration is suitable for a quick upgrade to an existing design using any PIC microcontroller.

For new CAN bus–based designs it is easier to use a PIC microcontroller with a built-in CAN module. As shown in Figure 9.12, such devices include built-in CAN controller hardware on the chip. All that is required to make a CAN node is to add a CAN transceiver chip. Table 9.2 lists some of the PIC microcontrollers that include a CAN module.

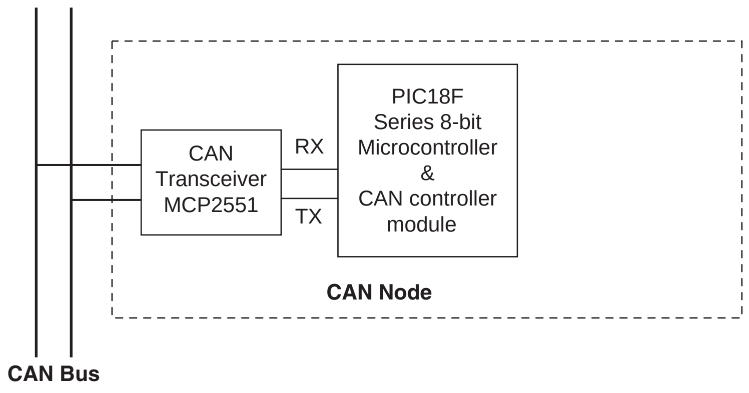


Figure 9.12: CAN node with integrated CAN module

Table 9.2: Some popular PIC microcontrollers that include CAN modules

Device	Pins	Flash (KB)	SRAM (KB)	EEPROM (bytes)	A/D	CAN module	SPI	UART
18F258	28	16	768	256	5	1	1	1
18F2580	28	32	1536	256	8	1	1	1
18F2680	28	64	3328	1024	8	1	1	1
18F4480	40/44	16	768	256	11	1	1	1
18F8585	80	48	3328	1024	16	1	1	1
18F8680	80	64	3328	1024	16	1	1	1

9.9 PIC18F258 Microcontroller

Later in this chapter the PIC18F258 microcontroller is used in a CAN bus–based project. This section describes this microcontroller and its operating principles with respect to its built-in CAN bus. The principles here are in general applicable to other PIC microcontrollers with CAN modules.

The PIC18F258 is a high performance 8-bit microcontroller with integrated CAN module. The device has the following features:

- 32K flash program memory
- 1536 bytes RAM data memory
- 256 bytes EEPROM memory
- 22 I/O ports
- 5-channel 10-bit A/D converters
- Three timers/counters
- Three external interrupt pins
- High-current (25mA) sink/source
- Capture/compare/PWM module
- SPI/I^{2}C module
- CAN 2.0A/B module
- Power-on reset and power-on timer
- Watchdog timer
- Priority level interrupts
- DC to 40MHz clock input
- 8 $\times$ 8 hardware multiplier
- Wide operating voltage (2.0V to 5.5V)
- Power-saving sleep mode

The features of the PIC18F258 microcontroller's CAN module are as follows:

- Compatible with CAN 1.2, CAN 2.0A, and CAN 2.0B
- Supports standard and extended data frames
- Programmable bit rate up to 1Mbit/s
- Double-buffered receiver
- Three transmit buffers
- Two receive buffers
- Programmable clock source
- Six acceptance filters
- Two acceptance filter masks
- Loop-back mode for self-testing
- Low-power sleep mode
- Interrupt capabilities

The CAN module uses port pins RB3/CANRX and RB2/CANTX for CAN bus receive and transmit functions respectively. These pins are connected to the CAN bus via an MCP2551-type CAN bus transceiver chip.

The PIC18F258 microcontroller supports the following frame types:

- Standard data frame
- Extended data frame
- Remote frame
- Error frame
- Overload frame
- Interframe space

A node uses filters to decide whether or not to accept a received message. Message filtering is applied to the whole identifier field, and mask registers are used to specify which bits in the identifier the filters should examine.

The CAN module in the PIC18F258 microcontroller has six modes of operation:

- Configuration mode
- Disable mode
- Normal operation mode
- Listen-only mode
- Loop-back mode
- Error recognition mode

9.9.1 Configuration Mode

The CAN module is initialized in configuration mode. The module is not allowed to enter configuration mode while a transmission is taking place. In configuration mode the module will neither transmit nor receive, the error counters are cleared, and the interrupt flags remain unchanged.

9.9.2 Disable Mode

In disable mode, the module will neither transmit nor receive. In this mode the internal clock is stopped unless the module is active. If the module is active, it will wait for 11 recessive bits on the CAN bus, detect that condition as an IDLE bus, and then accept the module disable command. The WAKIF interrupt (wake-up interrupt) is the only CAN module interrupt that is active in disable mode.

9.9.3 Normal Operation Mode

The normal operation mode is the CAN module's standard operating mode. In this mode, the module monitors all bus messages and generates acknowledge bits, error frames, etc. This is the only mode that can transmit messages.

9.9.4 Listen-only Mode

The listen-only mode allows the CAN module to receive messages, including messages with errors. It can be used to monitor bus activities or to detect the baud rate on the bus. For automatic baud rate detection, at least two other nodes must be communicating with each other. The baud rate can be determined by testing different values until valid messages are received. The listen-only mode cannot transmit messages.

9.9.5 Loop-Back Mode

In the loop-back mode, messages can be directed from internal transmit buffers to receive buffers without actually transmitting messages on the CAN bus. This mode is useful during system developing and testing.

9.9.6 Error Recognition Mode

The error recognition mode is used to ignore all errors and receive all messages. In this mode, all messages, valid or invalid are received and copied to the receive buffer.

9.9.7 CAN Message Transmission

The PIC18F258 microcontroller implements three dedicated transmit buffers: TXB0, TXB1, and TXB2. Pending transmittable messages are in a priority queue. Before the SOF is sent, the priorities of all buffers queued for transmission are compared. The transmit buffer with the highest priority is sent first. If two buffers have the same priority, the one with the higher buffer number is sent first. There are four levels of priority.

9.9.8 CAN Message Reception

Reception of a message is a more complex process. The PIC18F258 microcontroller includes two receive buffers, RXB0 and RXB1, with multiple acceptance filters for each (see Figure 9.13). All received messages are assembled in the message assembly buffer (MAB). Once a message is received, regardless of the type of identifier and the number of data bytes, the entire message is copied into the MAB.

Received messages have priorities. RXB0 is the higher priority buffer, and it has two message acceptance filters, RXF0 and RXF1. RXB1 is the lower priority buffer and has four acceptance filters: RXF2, RXF3, RXF4, and RXF5. Two programmable acceptance filter masks, RXM0 and RXM1, are also available, one for each receive buffer.

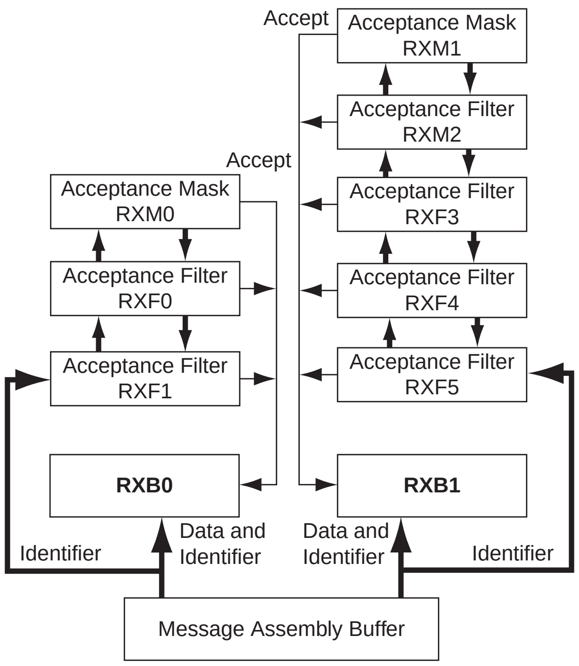


Figure 9.13: Receive buffer block diagram

The CAN module uses message acceptance filters and masks to determine if a message in the MAB should be loaded into a receive buffer. Once a valid message is received by the MAB, the identifier field of the message is compared to the filter values. If there is a match, that message is loaded into the appropriate receive buffer. The filter masks determine which bits in the identifier are examined with the filters. The truth table in Table 9.3 shows how each bit in the identifier is compared against

Table 9.3: Filter/mask truth table

Mask bit n	Filter bit n	Message identifier bit n001	Accept or reject bit n
0	×	×	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

the masks and filters to determine if the message should be accepted. If a mask bit is set to 0, that bit in the identifier is automatically accepted regardless of the filter bit.

9.9.9 Calculating the Timing Parameters

Setting the nodes' timing parameters is essential for the bus to operate reliably. Given the microcontroller clock frequency and the required CAN bus bit rate, we can calculate the values of the following timing parameters:

- Baud rate prescaler value
- Prop_Seg value
- Phase_Seg1 value
- Phase_Seg2 value
- SJW value

Correct timing requires that

- Prop_Seg + Phase_Seg1 $\geq$ Phase_Seg2
- Phase_Seg2 $\geq$ SJW

The following example illustrates the calculation of these timing parameters.

Example 9.2

Assuming the microcontroller oscillator clock rate is 20MHz and the required CAN bit rate is 125KHz, calculate the timing parameters.

Solution 9.2

With a 20MHz clock rate, the clock period is 50ns. Choosing a baud rate prescaler value of 4, from Equation (9.4), $T_Q = 2 * (BRP + 1) * T_{OSC}$, gives a time quantum of $T_Q = 500$ns. To obtain a nominal bit rate of 125KHz, the nominal bit time must be:

$$T_{BIT} = 1/0.125\text{MHz} = 8\mu\text{s, or } 16T_Q$$

The *Sync_Segment* is $1T_Q$. Choosing $2T_Q$ for the *Prop_Seg*, and $7T_Q$ for *Phase_Seg1* leaves $6T_Q$ for *Phase_Seg2* and places the sampling point at $10T_Q$ at the end of *Phase_Seg1*.

By the rules described earlier, the SJW can be the maximum allowed (i.e., 4). However, a large SJW is only necessary when the clock generation of different nodes is not stable or accurate (e.g., if ceramic resonators are used). Typically, a SJW of 1 is enough. In summary, the required timing parameters are:

```
Baud rate prescaler (BRP) = 4
Sync_Seg                  = 1
Prop_Seg                  = 2
Phase_Seg1                = 7
Phase_Seg2                = 6
SJW                       = 1
```

The sampling point is at $10T_Q$ which corresponds to 62.5% of the total bit time.

There are several tools available for free on the Internet for calculating CAN bus timing parameters. One such tool is the CAN Baud Rate Calculator, developed by Artic Consultants Ltd (http://www.articconsultants.co.uk). An example using this tool follows.

Example 9.3

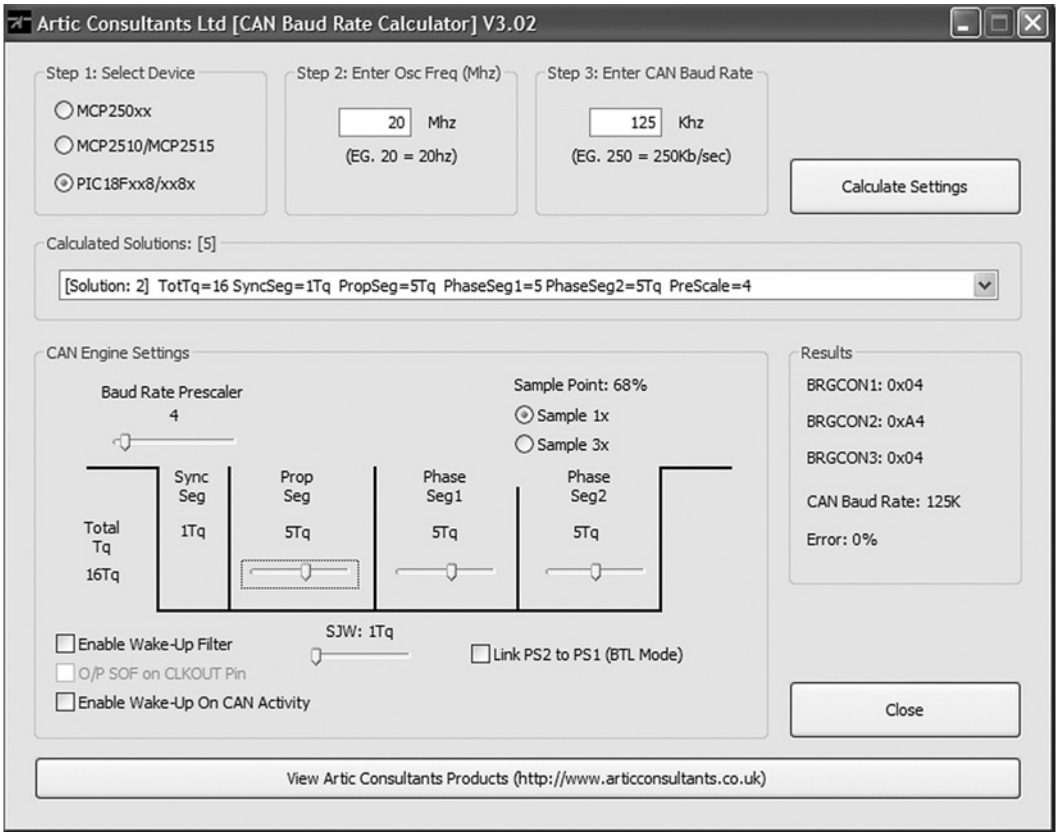
Assuming the microcontroller oscillator clock rate is 20MHz and the required CAN bit rate is 125KHz, calculate the timing parameters using the CAN Baud Rate Calculator.

Solution 9.3

Figure 9.14 shows the output of the CAN Baud Rate Calculator program. The device type is selected as PIC18Fxxx8, the oscillator frequency is entered as 20MHz, and the CAN bus baud rate is entered as 125KHz.

Clicking the Calculate Settings button calculates and displays the recommended timing parameters. In general, there is more than one solution, and different solutions are given in the Calculated Solutions field's drop-down menu.

In choosing Solution 2 from the drop-down menu, the following timing parameters are recommended by the program:



Artic Consultants Ltd [CAN Baud Rate Calculator] V3.02

Step 1: Select Device

MCP250xx

MCP2510/MCP2515

[✓ PIC18Fxx8/xx8x]

Step 2: Enter Osc Freq (Mhz)

20 Mhz

(EG. 20 = 20hz)

Step 3: Enter CAN Baud Rate

125 Khz

(EG. 250 = 250Kb/sec)

Calculated Solutions: [5]

[Solution: 2] TotTq=16 SyncSeg=1Tq PropSeg=5Tq PhaseSeg1=5 PhaseSeg2=5Tq PreScale=4

CAN Engine Settings

Baud Rate Prescaler 4

Sample Point: 68%

[✓ Sample 1x]

Sample 3x

Total Tq 16Tq

Sync Seg 1Tq

Prop Seg 5Tq

Phase Seg1 5Tq

Phase Seg2 5Tq

SJW: 1Tq

Enable Wake-Up Filter

O/P SOF on CLKOUT Pin

Enable Wake-Up On CAN Activity

Link PS2 to PS1 (BTL Mode)

Results

BRGCON1: 0x04

BRGCON2: 0xA4

BRGCON3: 0x04

CAN Baud Rate: 125K

Error: 0%

Calculate Settings
Close
View Artic Consultants Products (http://www.articconsultants.co.uk)

Figure 9.14: Output of the CAN Baud Rate Calculator program

```
Baud rate prescaler (BRP) = 4
Sync_Seg                  = 1
Prop_Seg                  = 5
Phase_Seg1                = 5
Phase_Seg2                = 5
SJW                       = 1
Sample point              = 68%
Error                     = 0%
```

9.10 mikroC CAN Functions

The mikroC language provides two libraries for CAN bus applications: the library for PIC microcontrollers with built-in CAN modules and the library based on using a SPI bus for PIC microcontrollers having no built-in CAN modules. In this section we will discuss only the library functions available for PIC microcontrollers with built-in CAN modules. Similar functions are available for the PIC microcontrollers with no built-in CAN modules.

The mikroC CAN functions are supported only by PIC18XXX8 microcontrollers with MCP2551 or similar CAN transceivers. Both standard (11 identifier bits) and extended format (29 identifier bits) messages are supported.

The following mikroC functions are provided:

- CANSetOperationMode
- CANGetOperationMode
- CANInitialize
- CANSetBaudRAte
- CANSetMask
- CANSetFilter
- CANRead
- CANWrite

9.10.1 CANSetOperationMode

The *CANSetOperationMode* function sets the CAN operation mode. The function prototype is:

```
void CANSetOperationMode(char mode, char wait_flag)
```

The parameter *wait_ flag* is either 0 or 0 × FF. If it is set to 0 × FF, the function blocks and will not return until the requested mode is set. If it is set to 0, the function returns as a nonblocking call.

The mode can be one of the following:

• CAN_MODE_NORMAL	Normal mode of operation
• CAN_MODE_SLEEP	Sleep mode of operation
• CAN_MODE_LOOP	Loop-back mode of operation

• CAN_MODE_LISTEN	Listen-only mode of operation
• CAN_MODE_CONFIG	Configuration mode of operation

9.10.2 CANGetOperationMode

The *CANGetOperationMode* function returns the current CAN operation mode. The function prototype is:

```
char CANGetOperationMode(void)
```

9.10.3 CANInitialize

The *CANInitialize* function initializes the CAN module. All mask registers are cleared to 0 to allow all messages. Upon execution of this function, the normal mode is set. The function prototype is:

```
void CANInitialize(char SJW, char BRP, char PHSEG1, char PHSEG2,
char PROPEG, char CAN_CONFIG_FLAGS)
```

where

SJW	is the synchronization jump width
BRP	is the baud rate prescaler
PHSEG1	is the Phase_Seg1 timing parameter
PHSEG2	is the Phase_Seg2 timing parameter
PROPSEG	is the Prop_Seg

CAN_CONFIG_FLAGS can be one of the following configuration flags:

• CAN_CONFIG_DEFAULT	Default flags
• CAN_CONFIG_PHSEG2_PRG_ON	Use supplied PHSEG2 value
• CAN_CONFIG_PHSEG2_PRG_OFF	Use maximum of PHSEG1 or information processing time (IPT), whichever is greater
• CAN_CONFIG_LINE_FILTER_ON	Use CAN bus line filter for wake-up
• CAN_CONFIG_FILTER_OFF	Do not use CAN bus line filter

• CAN_CONFIG_SAMPLE_ONCE	Sample bus once at sample point
• CAN_CONFIG_SAMPLE_THRICE	Sample bus three times prior to sample point
• CAN_CONFIG_STD_MSG	Accept only standard identifier messages
• CAN_CONFIG_XTD_MSG	Accept only extended identifier messages
• CAN_CONFIG_DBL_BUFFER_ON	Use double buffering to receive data
• CAN_CONFIG_DBL_BUFFER_OFF	Do not use double buffering
• CAN_CONFIG_ALL_MSG	Accept all messages including invalid ones
• CAN_CONFIG_VALID_XTD_MSG	Accept only valid extended identifier messages
• CAN_CONFIG_VALID_STD_MSG	Accept only valid standard identifier messages
• CAN_CONFIG_ALL_VALID_MSG	Accept all valid messages

These configuration values can be bitwise AND'ed to form complex configuration values.

9.10.4 CANSetBaudRate

The *CANSetBaudRate* function is used to set the CAN bus baud rate. The function prototype is:

```
void CANSetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2,
char PROPSEG, char CAN_CONFIG_FLAGS)
```

The arguments of the function are as in function *CANInitialize*.

9.10.5 CANSetMask

The *CANSetMask* function sets the mask for filtering messages. The function prototype is:

```
void CANSetMask(char CAN_MASK, long value, char
CAN_CONFIGFLAGS)
```

CAN_MASK can be one of the following:

• CAN_MASK_B1	Receive buffer 1 mask value
• CAN_MASK_B2	Receive buffer 2 mask value

value is the mask register value. CAN_CONFIG_FLAGS can be either CAN_CONFIG_XTD (extended message), or CAN_CONFIG_STD (standard message).

9.10.6 CANSetFilter

The *CANSetFilter* function sets filter values. The function prototype is:

```
void CANSetFilter(char CAN_FILTER, long value, char
CAN_CONFIG_FLAGS)
```

CAN_FILTER can be one of the following:

• CAN_FILTER_B1_F1	Filter 1 for buffer 1
• CAN_FILTER_B1_F2	Filter 2 for buffer 1
• CAN_FILTER_B2_F1	Filter 1 for buffer 2
• CAN_FILTER_B2_F2	Filter 2 for buffer 2
• CAN_FILTER_B2_F3	Filter 3 for buffer 2
• CAN_FILTER_B2_F4	Filter 4 for buffer 2

CAN_CONFIG_FLAGS can be either CAN_CONFIG_XTD (extended message) or CAN_CONFIG_STD (standard message).

9.10.7 CANRead

The *CANRead* function is used to read messages from the CAN bus. If no message is available, 0 is returned. The function prototype is:

```
char CANRead(long *id, char *data, char *datalen, char
*CAN_RX_MSG_FLAGS)
```

id is the CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended). *data* is an array of bytes up to 8 where the received data is stored. *datalen* is the length of the received data (1 to 8).

CAN_RX_MSG_FLAGS can be one of the following:

• CAN_RX_FILTER_1	Receive buffer filter 1 accepted this message
• CAN_RX_FILTER_2	Receive buffer filter 2 accepted this message
• CAN_RX_FILTER_3	Receive buffer filter 3 accepted this message
• CAN_RX_FILTER_4	Receive buffer filter 4 accepted this message
• CAN_RX_FILTER_5	Receive buffer filter 5 accepted this message
• CAN_RX_FILTER_6	Receive buffer filter 6 accepted this message
• CAN_RX_OVERFLOW	Receive buffer overflow occurred
• CAN_RX_INVALID_MSG	Invalid message received
• CAN_RX_XTD_FRAME	Extended identifier message received
• CAN_RX_RTR_FRAME	RTR frame message received
• CAN_RX_DBL_BUFFERED	This message was double buffered

These flags can be bitwise AND'ed if desired.

9.10.8 CANWrite

The *CANWrite* function is used to send a message to the CAN bus. A zero is returned if message can not be queued (buffer full). The function prototype is:

```
char CANWrite(long id, char *data, char datalen, char
CAN_TX_MSG_FLAGS)
```

id is the CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended). *data* is an array of bytes up to 8 where the data to be sent is stored. *datalen* is the length of the data (1 to 8).

CAN_TX_MSG_FLAGS can be one of the following:

• CAN_TX_PRIORITY_0	Transmit priority 0
• CAN_TX_PRIORITY_1	Transmit priority 1

• CAN_TX_PRIORITY_2	Transmit priority 2
• CAN_TX_PRIORITY_3	Transmit priority 3
• CAN_TX_STD_FRAME	Standard identifier message
• CAN_TX_XTD_FRAME	Extended identifier message
• CAN_TX_NO_RTR_FRAME	Non RTR message
• CAN_TX_RTR_FRAME	RTR message

These flags can be bitwise AND'ed if desired.

9.11 CAN Bus Programming

To operate the PIC18F258 microcontroller on the CAN bus, perform the following steps:

- Configure the CAN bus I/O port directions (RB2 and RB3)
- Initialize the CAN module (*CANInitialize*)
- Set the CAN module to CONFIG mode (*CANSetOperationMode*)
- Set the mask registers (*CANSetMask*)
- Set the filter registers (*CANSetFilter*)
- Set the CAN module to normal mode (*CANSetOperationMode*)
- Write/read data (*CANWrite*/*CANRead*)

PROJECT 9.1—Temperature Sensor CAN Bus Project

The following is a simple two-node CAN bus–based project. The block diagram of the project is shown in Figure 9.15. The system is made up of two CAN nodes. One node (called DISPLAY node) requests the temperature every second and displays it on an LCD. This process is repeated continuously. The other node (called COLLECTOR node) reads the temperature from an external semiconductor temperature sensor.

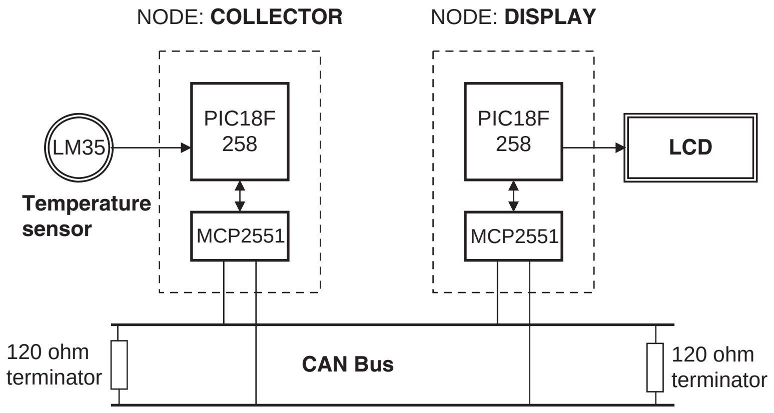


Figure 9.15: Block diagram of the project

The project’s circuit diagram is given in Figure 9.16. Two CAN nodes are connected together using a two-meter twisted pair cable, terminated with a 120-ohm resistor at each end.

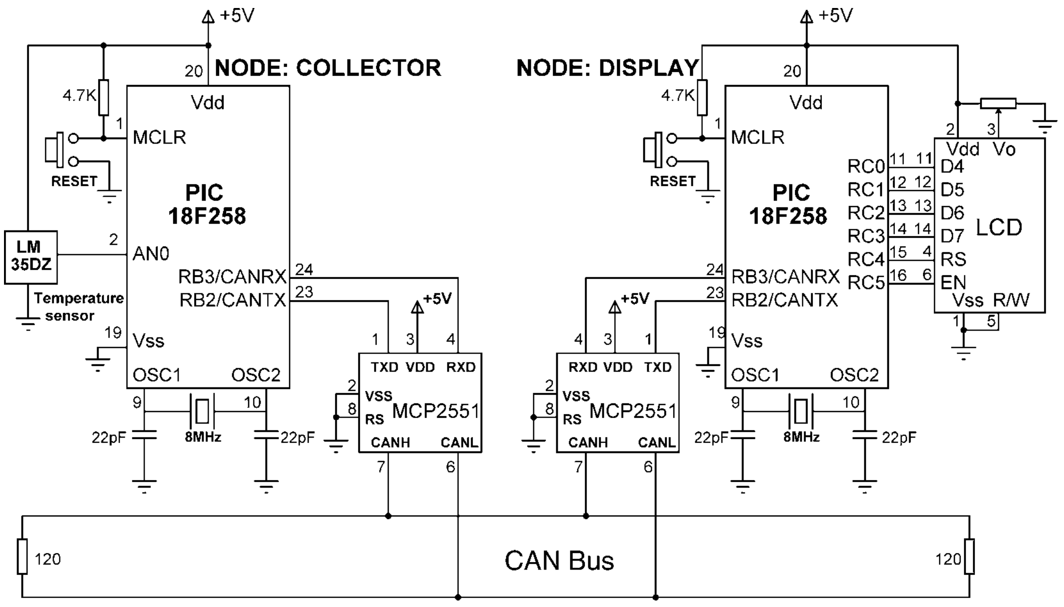


Figure 9.16: Circuit diagram of the project

The DISPLAY Processor

Like the COLLECTOR processor, the DISPLAY processor consists of a PIC18F258 microcontroller with a built-in CAN module and an MCP2551 transceiver chip. The microcontroller is operated from an 8MHz crystal. The MCLR input is connected to an external reset button. The CAN outputs (RB2/CANTX and RB3/CANRX) of the microcontroller are connected to the TXD and RXD inputs of the MCP2551. Pins CANH and CANL of the transceiver chip are connected to the CAN bus. An HD44780-type LCD is connected to PORTC of the microcontroller to display the temperature values.

The COLLECTOR Processor

The COLLECTOR processor consists of a PIC18F258 microcontroller with a built-in CAN module and an MCP2551 transceiver chip. The microcontroller is operated from an 8MHz crystal. The MCLR input is connected to an external reset button. Analog input AN0 of the microcontroller is connected to a LM35DZ-type semiconductor temperature sensor. The sensor can measure temperature in the range of 0°C to 100°C and generates an analog voltage directly proportional to the measured temperature (i.e., the output is 10mV/°C). For example, at 20°C the output voltage is 200mV.

The CAN outputs (RB2/CANTX and RB3/CANRX) of the microcontroller are connected to the TXD and RXD inputs of an MCP2551-type CAN transceiver chip. The CANH and CANL outputs of this chip are connected directly to a twisted cable terminating at the CAN bus. The MCP2551 is an 8-pin chip that supports data rates up to 1Mb/s. The chip can drive up to 112 nodes. An external resistor connected to pin 8 of the chip controls the rise and fall times of CANH and CANL so that EMI can be reduced. For high-speed operation this pin should be connected to ground. A reference voltage equal to VDD/2 is output from pin 5 of the chip.

The program listing is in two parts: the DISPLAY program and the COLLECTOR program. The operation of the system is as follows:

- The DISPLAY processor requests the current temperature from the COLLECTOR processor over the CAN bus

- The COLLECTOR processor reads the temperature, formats it, and sends to the DISPLAY processor over the CAN bus
- The DISPLAY processor reads the temperature from the CAN bus and then displays it on the LCD
- This process is repeated every second

DISPLAY Program

Figure 9.17 shows the program listing of the DISPLAY program, called DISPLAY.C. At the beginning of the program PORTC pins are configured as outputs, RB3 is configured as input (CANRX), and RB2 is configured as output (CANTX). In this project the CAN bus bit rate is selected as 100Kb/s. With a microcontroller clock frequency of 8MHz, the Baud Rate Calculator program (see Figure 9.14) is used to calculate the timing parameters as:

```
SJW = 1
BRP = 1
Phase_Seg1 = 6
Phase_Seg2 = 7
Prop_Seg = 6
```

The mikroC CAN bus function *CANInitialize* is used to initialize the CAN module. The timing parameters and the initialization flag are specified as arguments in this function. The initialization flag is made up from the bitwise AND of:

```
init_flag = CAN_CONFIG_SAMPLE_THRICE &
            CAN_CONFIG_PHSEG2_PRG_ON &
            CAN_CONFIG_STD_MSG &
            CAN_CONFIG_DBL_BUFFER_ON &
            CAN_CONFIG_VALID_XTD_MSG &
            CAN_CONFIG_LINE_FILTER_OFF;
```

Where sampling the bus three times is specified, the standard identifier is specified, double buffering is turned on, and the line filter is turned off.

Then the operation mode is set to CONFIG and the filter masks and filter values are specified. Both mask 1 and mask 2 are set to all 1's (−1 is a shorthand way of writing hexadecimal FFFFFFFF, i.e., setting all mask bits to 1's) so that all filter bits match up with incoming data.

```
/***************************************************************************
                CAN BUS EXAMPLE - NODE: DISPLAY
                ==============================

This is the DISPLAY node of the CAN bus example. In this project a PIC18F258
type microcontroller is used. An MCP2551 type CAN bus transceiver is used to
connect the microcontroller to the CAN bus. The microcontroller is operated from
an 8MHz crystal with an external reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins RXD
and TXD of the transceiver chip respectively. Pins CANH and CANL of
the transceiver chip are connected to the CAN bus.

An LCD is connected to PORTC of the microcontroller. The ambient
temperature is read from another CAN node and is displayed on the LCD.

The LCD is connected to the microcontroller as follows:

Microcontroller     LCD

   RC0           D4
   RC1           D5
   RC2           D6
   RC3           D7
   RC4           RS
   RC5           EN

CAN speed parameters are:

   Microcontroller clock:       8MHz
   CAN Bus bit rate:            100Kb/s
   Sync_Seg:                    1
   Prop_Seg:                    6
   Phase_Seg1:                  6
   Phase_Seg2:                  7
   SJW:                         1
   BRP:                         1
   Sample point:                65%

Author:         Dogan Ibrahim
Date:           October 2007
File:           DISPLAY.C
***************************************************************************/

void main()
{
    unsigned char temperature, data[8];
    unsigned short init_flag, send_flag, dt, len, read_flag;
    char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4];
    long id, mask;
```

Figure 9.17: DISPLAY program listing

```
     TRISC = 0;                             // PORTC are outputs (LCD)
     TRISB = 0x08;                          // RB2 is output, RB3 is input
//
// CAN BUS Parameters
//
     SJW = 1;
     BRP = 1;
     Phase_Seg1 = 6;
     Phase_Seg2 = 7;
     Prop_Seg = 6;

     init_flag = CAN_CONFIG_SAMPLE_THRICE  &
                 CAN_CONFIG_PHSEG2_PRG_ON  &
                 CAN_CONFIG_STD_MSG        &
                 CAN_CONFIG_DBL_BUFFER_ON  &
                 CAN_CONFIG_VALID_XTD_MSG  &
                 CAN_CONFIG_LINE_FILTER_OFF;

     send_flag = CAN_TX_PRIORITY_0         &
                 CAN_TX_XTD_FRAME          &
                 CAN_TX_NO_RTR_FRAME;

     read_flag = 0;
//
// Initialize CAN module
//
      CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, init_flag);
//
// Set CAN CONFIG mode
//
      CANSetOperationMode(CAN_MODE_CONFIG, 0xFF);

      mask = -1;
//
// Set all MASK1 bits to 1's
//
      CANSetMask(CAN_MASK_B1, mask, CAN_CONFIG_XTD_MSG);
//
// Set all MASK2 bits to 1's
//
      CANSetMask(CAN_MASK_B2, mask, CAN_CONFIG_XTD_MSG);
//
// Set id of filter B2_F3 to 3
//
      CANSetFilter(CAN_FILTER_B2_F3,3,CAN_CONFIG_XTD_MSG);
//
// Set CAN module to NORMAL mode
//
      CANSetOperationMode(CAN_MODE_NORMAL, 0xFF);
```

Figure 9.17: (Cont'd)

```
//
// Configure LCD
//
    Lcd_Config(&PORTC,4,5,0,3,2,1,0);          // LCD is connected to PORTC
    Lcd_Cmd(LCD_CLEAR);                        // Clear LCD
    Lcd_Out(1,1,"CAN BUS");                    // Display heading on LCD
    Delay_ms(1000);                            // Wait for 2 seconds

//
// Program loop. Read the temperature from Node:COLLECTOR and display
// on the LCD continuously
//
    for(;;)                                    // Endless loop
    {
      Lcd_Cmd(LCD_CLEAR);                      // Clear LCD
      Lcd_Out(1,1,"Temp = ");                  // Display "Temp = "
      //
      // Send a message to Node:COLLECTOR and ask for data
      //
      data[0] = 'T';                           // Data to be sent
      id = 500;                                // Identifier
      CANWrite(id, data, 1, send_flag);        // send 'T'
      //
      // Get temperature from node:COLLECT
      //
      dt = 0;
      while(!dt)dt = CANRead(&id, data, &len, &read_flag);
      if(id == 3)
      {
        temperature = data[0];
        ByteToStr(temperature,txt);            // Convert to string
        Lcd_Out(1,8,txt);                      // Output to LCD
        Delay_ms(1000);                        // Wait 1 second
      }
    }

}
```

Figure 9.17: (Cont'd)

Filter 3 for buffer 2 is set to value 3 so that identifiers having values 3 are accepted by the receive buffer.

The operation mode is then set to NORMAL. The program then configures the LCD and displays the message “CAN BUS” for one second on the LCD.

The main program loop executes continuously and starts with a *for* statement. Inside this loop the LCD is cleared and text “TEMP =” is displayed on the LCD. Then character “T” is sent over the bus with the identifier equal to 500 (the COLLECTOR

```
/*************************************************************************
                    CAN BUS EXAMPLE - NODE: COLLECTOR
                    ================================

This is the COLLECTOR node of the CAN bus example. In this project a
PIC18F258 type microcontroller is used. An MCP2551 type CAN bus transceiver
is used to connect the microcontroller to the CAN bus. The microcontroller is
operated from an 8MHz crystal with an external reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins RXD
and TXD of the transceiver chip respectively. Pins CANH and CANL of the
transceiver chip are connected to the CAN bus.

An LM35DZ type analog temperature sensor is connected to port AN0 of the
microcontroller. The microcontroller reads the temperature when a request is
received and then sends the temperature value as a byte to Node:DISPLAY on
the CAN bus.

CAN speed parameters are:

   Microcontroller clock:      8MHz
   CAN Bus bit rate:           100Kb/s
   Sync_Seg:                   1
   Prop_Seg:                   6
   Phase_Seg1:                 6
   Phase_Seg2:                 7
   SJW:                        1
   BRP:                        1
   Sample point:               65%

Author:        Dogan Ibrahim
Date:          October 2007
File:          COLLECTOR.C
*************************************************************************/

void main()
{
    unsigned char temperature, data[8];
    unsigned short init_flag, send_flag, dt, len, read_flag;
    char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4];
    unsigned int temp;
    unsigned long mV;
    long id, mask;

    TRISA = 0xFF;                       // PORTA are inputs
    TRISB = 0x08;                       // RB2 is output, RB3 is input
//
// Configure A/D converter
//
    ADCON1 = 0x80;
```

Figure 9.18: COLLECTOR program listing

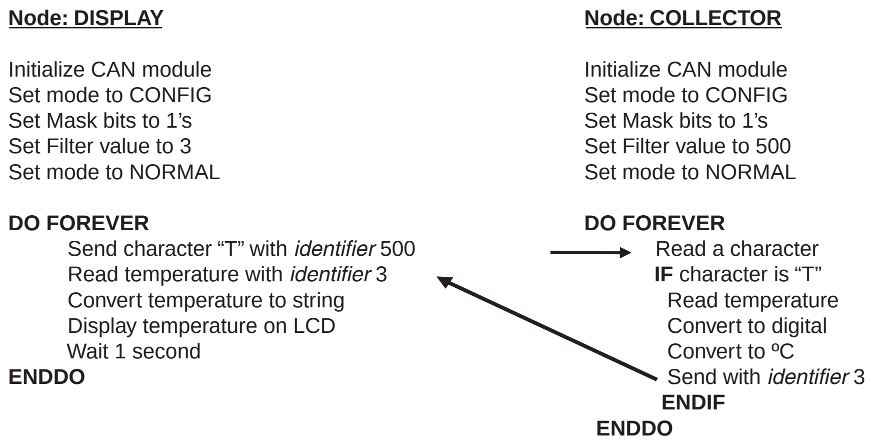
(Continued)

```
//
// CAN BUS Timing Parameters
//
   SJW = 1;
   BRP = 1;
   Phase_Seg1 = 6;
   Phase_Seg2 = 7;
   BRP = 1;
   Prop_Seg = 6;

   init_flag = CAN_CONFIG_SAMPLE_THRICE  &
            CAN_CONFIG_PHSEG2_PRG_ON  &
            CAN_CONFIG_STD_MSG       &
            CAN_CONFIG_DBL_BUFFER_ON  &
            CAN_CONFIG_VALID_XTD_MSG  &
            CAN_CONFIG_LINE_FILTER_OFF;

   send_flag = CAN_TX_PRIORITY_0         &
            CAN_TX_XTD_FRAME          &
            CAN_TX_NO_RTR_FRAME;

   read_flag = 0;
//
// Initialise CAN module
//
    CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, init_flag);
//
// Set CAN CONFIG mode
//
    CANSetOperationMode(CAN_MODE_CONFIG, 0xFF);

    mask = -1;
//
// Set all MASK1 bits to 1's
//
    CANSetMask(CAN_MASK_B1, mask, CAN_CONFIG_XTD_MSG);
//
// Set all MASK2 bits to 1's
//
    CANSetMask(CAN_MASK_B2, mask, CAN_CONFIG_XTD_MSG);
//
// Set id of filter B1_F1 to 3
//
    CANSetFilter(CAN_FILTER_B2_F3,500,CAN_CONFIG_XTD_MSG);
//
// Set CAN module to NORMAL mode
//
    CANSetOperationMode(CAN_MODE_NORMAL, 0xFF);

//
```

Figure 9.18: (Cont'd)

```
// Program loop. Read the temperature from analog temperature
// sensor
//
   for(;;)                                                   // Endless loop
   {
     //
     // Wait until a request is received
     //
     dt = 0;
     while(!dt) dt = CANRead(&id, data, &len, &read_flag);
     if(id == 500 && data[0] == 'T')
     {
        //
        // Now read the temperature
        //
        temp = Adc_Read(0);                                  // Read temp
        mV = (unsigned long)temp * 5000 / 1024;              // in mV
        temperature = mV/10;                                 // in degrees C
        //
        // send the temperature to Node:Display
        //
        data[0] = temperature;
        id = 3;                                              // Identifier
        CANWrite(id, data, 1, send_flag);                    // send temperature
     }
    }
}
```

Figure 9.18: (Cont'd)



```
Node: DISPLAY

Initialize CAN module
Set mode to CONFIG
Set Mask bits to 1's
Set Filter value to 3
Set mode to NORMAL

DO FOREVER
     Send character "T" with identifier 500
     Read temperature with identifier 3
     Convert temperature to string
     Display temperature on LCD
     Wait 1 second
ENDDO
```

```
Node: COLLECTOR

Initialize CAN module
Set mode to CONFIG
Set Mask bits to 1's
Set Filter value to 500
Set mode to NORMAL

DO FOREVER
     Read a character
     IF character is "T"
       Read temperature
       Convert to digital
       Convert to ºC
       Send with identifier 3
     ENDIF
ENDDO
```

Figure 9.19: Operation of both nodes

node filter is set to accept identifier 500). This is a request to the COLLECTOR node to send the temperature reading. The program then reads the temperature from the CAN bus, converts it to a string in array *txt*, and displays it on the LCD. This process repeats after a one-second delay.

COLLECTOR Program

Figure 9.18 shows the program listing of the COLLECTOR program, called COLLECTOR.C. The initial part of this program is the same as the DISPLAY program. The receive filter is set to 500 so that messages with identifier 500 are accepted by the program.

Inside the program loop, the program waits until it receives a request to send the temperature. Here the request is identified by the reception of character "T". Once a valid request is received, the temperature is read and converted into °C (stored in variable *temperature*) and then sent to the CAN bus as a byte with an identifier value equal to 3. This process repeats forever.

Figure 9.19 summarizes the operation of both nodes.

CHAPTER 10

Multi-Tasking and Real-Time Operating Systems

Nearly all microcontroller-based systems perform more than one activity. For example, a temperature monitoring system is made up of three tasks that normally repeat after a short delay, namely:

- Task 1 Reads the temperature
- Task 2 Formats the temperature
- Task 3 Displays the temperature

More complex systems may have many complex tasks. In a multi-tasking system, numerous tasks require CPU time, and since there is only one CPU, some form of organization and coordination is needed so each task has the CPU time it needs. In practice, each task takes a very brief amount of time, so it seems as if all the tasks are executing in parallel and simultaneously.

Almost all microcontroller-based systems work in real time. A real-time system is a time responsive system that can respond to its environment in the shortest possible time. Real time does not necessarily mean the microcontroller should operate at high speed. What is important in a real-time system is a fast response time, although high speed can help. For example, a real-time microcontroller-based system with various external switches is expected to respond immediately when a switch is activated or some other event occurs.

A real-time operating system (RTOS) is a piece of code (usually called the kernel) that controls task allocation when the microcontroller is operating in a multi-tasking

EMBEDDED ETHERNET:

Exchanging messages using UDP and TCP
Serving web pages with Dynamic Data
Serving web pages that respond to user Input
Email for Embedded Systems
Using FTP
Keeping Devices and Network secure.

7

Serving Web Pages that Respond to User Input

Chapter 6 showed how a Web page can use HTML to display text and images, including real-time data. Many embedded Web servers also need to display pages that can respond to user input. For example, a Web page might display a virtual control panel that enables users to start, stop, or modify processes controlled by an embedded system. Or a page might display a form that enables users to enter or select values for use in configuring or controlling a device.

‘Two technologies for enabling Web pages to respond to user input are common gateway interface (CGI) programming and Java servlets. CGI programs and Java servlets can do the following:

- Retrieve the current values of variables and place them on a Web page to return to a client.
- Receive and act on data provided by a client who clicks a hyperlink or submits an HTML form.

- Do just about anything that an ordinary program is capable of, including making calculations, performing logical operations, and accessing I/O ports.

This chapter presents examples of devices that use CGI programming and Java servlets to enable Web pages to respond to user input. A Rabbit module uses CGI programming and a TINI uses Java servlets. The In Depth section of the chapter has more detail about what's involved in using these technologies, plus examples of how a server can serve forms and respond to form data submitted by users.

Quick Start: Device Controller

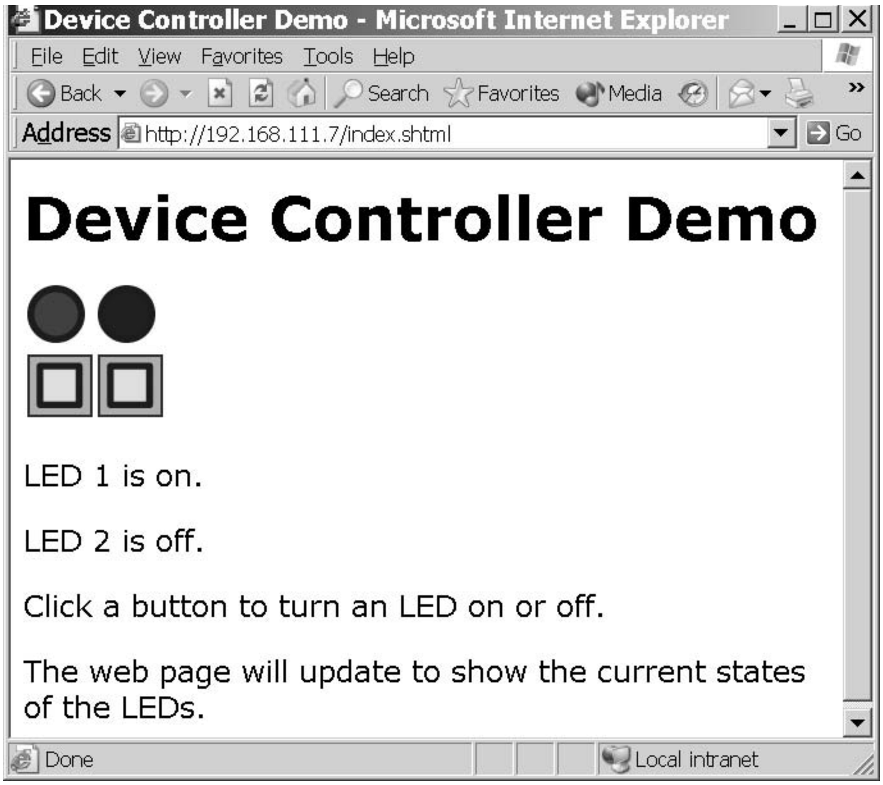
What method to use to enable a Web page to accept user input depends in part on the programming language and the system's capabilities. For the Rabbit, the HTTP server in Dynamic C's *http.lib* library supports Common Gateway Interface (CGI) programming. For the TINI, the addition of a servlet engine enables running servlets written in Java.

The browsers that display pages that request user input don't require any special capabilities. At the browser, a link or button that requests a server to take an action is just like any other hyperlink. A Web page that accepts user input on a form must support forms, but it's rare to find a browser these days without support for forms. On receiving the user's input, the server performs the processing and returns a Web page that may incorporate output from the program code the server has just executed.

The Device Controller's Web Page

The device-controller examples in this section use LEDs to represent processes the system is controlling. The examples can serve as prototypes for embedded systems that accept user input from Web pages for any purpose.

In the examples, the servers host a Web page that displays a virtual control panel (Figure 7-1). The Web page displays two LEDs and two buttons that


Device Controller Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.7/index.shtml Go
Done
Local intranet

Device Controller Demo

LED 1 is on.

LED 2 is off.

Click a button to turn an LED on or off.

The web page will update to show the current states of the LEDs.

Figure 7-1: This Web page is a virtual control panel that enables users to turn LEDs on or off by clicking a button.

users can click to turn the LEDs on and off. Both the Rabbit and TINI can host this Web page, though they use different technologies to respond to the button clicks.

The images of LEDs on the Web page match the states of the LEDs in the embedded system at the time the browser requested the Web page. When a user viewing the page clicks a button, the Web server receives a message containing the name of a CGI function or servlet to execute. The server toggles the state of the selected LED and then either returns a Web page containing updated images and text or returns a code that advises the client to request an updated page.

Rabbit Device Controller

The first example uses the same RabbitCore RCM3200 module as the previous Rabbit examples. Listing 7-1 shows the HTML code for Figure 7-1's Web page.

On the page, the images of the two LEDs and their buttons are in table cells so that each button lines up below the LED it controls. Each button is a hyperlink. When a user viewing the page clicks a button, the server receives a message containing the text */led1toggle.cgi* or */led2toggle.cgi*. On the server, these file names are associated with CGI functions.

The page also uses SSI `#echo` directives, as described in Chapter 6, to display images of lit or unlit LEDs and text descriptions of the LEDs' states ("on" or "off").

The LEDs are controlled by bits 6 and 7 of Port G on the Rabbit 3000 CPU. The LEDS are included on Rabbit Semiconductor's prototyping board for the RCM3200.

Program Code

In the RCM3200, a Dynamic C program serves Figure 7-1's Web page and responds to button clicks that send HTTP requests to execute CGI functions. Much of the code is similar to the code in Chapter 6's Rabbit example.

Initial Defines and Declarations

As explained in Chapter 5, `TCPCONFIG` specifies a configuration that sets the IP address, netmask, and gateway IP address values:

```
#define TCPCONFIG 1
```

The CGI functions use the values in the `REDIRECTHOST` and `REDIRECTTO` constants to tell the client's browser what Web page to request to display the result of a button click. If the Rabbit is behind a router that uses NAT and if you want the Web page to be accessible beyond the local network, `REDIRECTHOST` must be the router's public IP address or domain name.

```
#define REDIRECTHOST    _PRIMARY_STATIC_IP
#define REDIRECTTO "http://" REDIRECTHOST "/index.shtml"
```

```
<html>
<head>
  <title>Device Controller Demo</title>
</head>

<body>

<h1>Device Controller Demo</h1>

<table>
  <tr>
    <td> <img SRC="<!--#echo var="led1_image" -->"> </td>
    <td> <img SRC="<!--#echo var="led2_image" -->"> </td>
  </tr>
  <tr>
    <td> <a href="/led1toggle.cgi"> <img src="button.gif"> </a>
 </td>
    <td> <a href="/led2toggle.cgi"> <img src="button.gif"> </a>
 </td>
  </tr>
</table>

<p>LED 1 is <!--#echo var="led1_state" --> .</p>
<p>LED 2 is <!--#echo var="led2_state" --> .</p>

<p>Click a button to turn an LED on or off.</p>
<p>The Web page will update to show the current states of the
 LEDs.</p>

</body>
</html>
```

Listing 7-1: This Web page contains links to CGI programs that the Rabbit executes before serving Figure 7-1's page.

The `#memmap xmem` directive causes all C functions not declared as root to be stored in extended memory. The `dcrtcp.lib` library supports TCP/IP. The `http.lib` library supports HTTP functions.

```
#memmap xmem
#use "dcrtcp.lib"
```

```
#use "http.lib"
```

The `#ximport` directive retrieves a file from the PC being used for project development, stores the file's length and contents in the Rabbit's extended memory, and associates a symbol with the file's address in memory. This application uses one Web page and three image files. You must replace the file paths with paths that are valid for the files in your development PC.

```
#ximport "c:/rabbitserver/index.shtml"    index_html
#ximport "c:/rabbitserver/ledon.gif"      ledon_gif
#ximport "c:/rabbitserver/ledoff.gif"     ledoff_gif
#ximport "c:/rabbitserver/button.gif"     button_gif
```

An `HttpType` structure specifies the handler to use with different file extensions. If the handler is `NULL`, the server uses the default handler, which sends the file's contents unaltered. For the Device Controller application, the Web page uses the *.shtml* handler because the page contains SSI directives. Requests for files with *.cgi* and *.gif* extensions use the default handler. The structure below also specifies the default handler for HTML files.

```
const HttpType http_types[] =
{
   { ".shtml", "text/html", shtml_handler},
   { ".html", "text/html", NULL},
   { ".cgi", "", NULL},
   { ".gif", "image/gif", NULL}
};
```

The strings `ledon_image` and `ledoff_image` store the names of files that contain images of lit and unlit LEDs (`"ledon.gif"` and `"ledoff.gif"`). The string variables `led1_image` and `led2_image` each contain a file name for the image that matches the corresponding LED's state. The string variables `led1_state` and `led2_state` hold the text `"on"` or `"off"` as appropriate, to match the state of an LED.

```
const char ledon_image[] = "ledon.gif";
const char ledoff_image[] = "ledoff.gif";

char led1_image[15];
char led2_image[15];

char led1_state[4];
char led2_state[4];
```

Controlling the LEDs

Each LED has a function (`led1toggle()` and `led2toggle()`) that executes when the server receives a message indicating that a user has clicked that LED's button on the Web page. When called, the function receives a pointer to an `HttpState` structure that contains information about the current connection and request.

This is the code for LED1:

```
int led1toggle(HttpState* state)
{
  if (BitRdPortI(PGDR, 6) == 0) {
    // When the bit is 0, the LED is on, so turn it off.
    BitWrPortI(PGDR, &PGDRShadow, 1, 6);
    strcpy(led1_image,ledoff_image);
    strcpy(led1_state, "off");
    }
  else {
    // When the bit is 1, the LED is off, so turn it on.
    BitWrPortI(PGDR, &PGDRShadow, 0, 6);
    strcpy(led1_image,ledon_image);
    strcpy(led1_state, "on");
    }

   cgi_redirectto(state,REDIRECTTO);
   return 0;
}
```

Writing zero to a port bit that controls an LED turns the LED on, and writing 1 to the port bit turns the LED off. The routine for LED1 reads the state of Port G, bit 6 and toggles the bit to the opposite state.

The `BitRdPortI()` function returns the value of a bit on one of the Rabbit's internal ports. PGDR is Port G's data register, and 6 is the number of the bit to read.

The `BitWrPortI()` function writes a value to a bit in one of the Rabbit's internal ports. Again, PGDR is Port G's data register. The second parameter, PGDRShadow, is a variable that functions as a shadow register that contains the last value written to the register. Shadow registers are useful for storing the most recently written values to write-only registers. Program code can then learn a bit's value by reading the bit in the corresponding Shadow register. Port G has read/write access, but the `BitWrPortI()` function requires a shadow register. The function's third and fourth parameters are the value to write (1) and the bit number to write to (6).

After writing to the port bit, the routine stores a file name (`"ledon.gif"` or `"ledoff.gif"`, as appropriate) in `led1_image`, and stores `"on"` or `"off"` as appropriate in `led1_state`.

The call to the `cgi_redirectto()` function tells the server to return an HTTP response containing a response code that advises the client to request the URL stored in `REDIRECTTO`. In this application, the `REDIRECTTO` URL is the same *index.shtml* page the browser displayed when the user clicked a button. The newly retrieved page will contain updated images and text that reflect the current values of the LEDs. The statement `return 0` must follow the call to `cgi_redirectto()`.

The routine for LED2 is the same as the routine for LED1, except that it references LED2's port bit and variables:

```
int led2toggle(HttpState* state)
{
  if (BitRdPortI(PGDR, 7) == 0) {
    // When the bit is 0, the LED is on, so turn it off.
    BitWrPortI(PGDR, &PGDRShadow, 1, 7);
     strcpy(led2_image,ledoff_image);
       strcpy(led2_state, "off");
     }
  else {
    // When the bit is 1, the LED is off, so turn it on.
    BitWrPortI(PGDR, &PGDRShadow, 0, 7);
    strcpy(led2_image,ledon_image);
     strcpy(led2_state, "on");
    }

   cgi_redirectto(state,REDIRECTTO);
   return 0;
}
```

Specifying What the Web Server Can Access

As in Chapter 6's example, an `HttpSpec` structure contains information about the files, variables, and functions the Web server can access. The Web page and three image files each have an `HTTPSPEC_FILE` entry in the structure. When a browser requests the file *index.shtml* or the server's default file (`"/"`), the server serves the Web page stored at `index_html`. The four string variables that hold file names and LED state information each have an `HTTPSPEC_VARIABLE` entry. Two `HTTPSPEC_FUNCTION` entries associate the names of the CGI programs (`/led1toggle.cgi` and `/led2toggle.cgi)` with pointers to the functions `led1toggle` and `led2toggle`.

```
const HttpSpec http_flashspec[] =
{
  { HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE, "/index.shtml", index_html, NULL, 0,
       NULL, NULL},
   { HTTPSPEC_FILE, "/ledon.gif", ledon_gif, NULL, 0,
       NULL, NULL},
   { HTTPSPEC_FILE, "/ledoff.gif", ledoff_gif, NULL, 0,
       NULL, NULL},
   { HTTPSPEC_FILE, "/button.gif", button_gif, NULL, 0,
       NULL, NULL},

   { HTTPSPEC_VARIABLE, "led1_image", 0, led1_image,
       PTR16, "%s", NULL},
   { HTTPSPEC_VARIABLE, "led2_image", 0, led2_image,
       PTR16, "%s", NULL},
   { HTTPSPEC_VARIABLE, "led1_state", 0, led1_state,
       PTR16,   "%s", NULL},
   { HTTPSPEC_VARIABLE, "led2_state", 0, led2_state,
       PTR16,   "%s", NULL},

   { HTTPSPEC_FUNCTION, "/led1toggle.cgi", 0, led1toggle,
       0, NULL, NULL},
   { HTTPSPEC_FUNCTION, "/led2toggle.cgi", 0, led2toggle,
       0, NULL, NULL},
};
```

The main() Function

The program's `main()` function begins by writing to Port G's Data Direction Register (PGDDR) to configure bits 6 and 7 as outputs. Writing 1 to a bit in the register configures the corresponding port bit as an output. The program then reads the bits and stores the appropriate file names and text to use on the Web page to reflect the LEDs' states.

```
main()
{
   WrPortI(PGDDR, NULL, 0xC0);

   if (BitRdPortI(PGDR, 6) == 0) {
     strcpy(led1_image,ledon_image);
     strcpy(led1_state, "on");
   } else {
     strcpy(led1_image,ledoff_image);
     strcpy(led1_state, "off");
   }

   if (BitRdPortI(PGDR, 7) == 0) {
     strcpy(led2_image,ledon_image);
     strcpy(led2_state, "on");
   } else {
     strcpy(led2_image,ledoff_image);
     strcpy(led2_state, "off");
   }
```

Before performing any network communications, the program must initialize the TCP/IP stack and Web server. As in Chapter 6's Rabbit example, calling `tcp_reserveport()` can improve the Web server's performance.

```
    sock_init();
    http_init();
    tcp_reserveport(80);
```

The main program loop has just one task, calling `http_handler()`. In a real-world application, the main program loop would perform other tasks as well.

```
    while (1) {
       http_handler();
       // Code to perform other tasks can be placed here.
    }
 } // end main
```

Using the Device Controller

When the RCM3200 module runs this program, any computer that can access the module over the network can request the Web page and view and control the LEDs. Clicking a button on the Web page causes the browser to send an HTTP request containing the name of a CGI function. The Rabbit executes the named function and returns a response code that advises the browser to refresh the page. In a similar way, you can enable users to control other processes on an RCM3200 or similar module via a Web page.

TINI Device Controller

To serve Figure 7-1's Device Controller Web page, a TINI can use Java servlets. Servlets are components that can place real-time data on a Web page and can receive and respond to user input, as well as performing just about any task that an ordinary program might do. A Web server that runs servlet code must have a servlet container, also called a servlet engine, which adds support for servlets to the Web server.

The servlet examples in this book are written for use with the Tynamo Web server, an HTTP server and servlet container from Shawn Silverman (*tynamo.qindesign.com*). Tynamo is free for development and educational use. Use in commercial products requires a license.

Another option for servlets on a TINI is Smart Software Consulting's TiniHttpServer (*www.smartsc.com*). TiniHttpServer is offered at no cost under the GNU General Public License. The source code is available.

For the latest information on licensing terms for both products, see their Web sites. The Web sites also have complete documentation, including links to the necessary files to download and instructions for building the servers and deploying them on a TINI module and other Java platforms.

The capabilities of any servlet engine will comply with the Java Servlet Specification, but the implementation details can vary with different products.

The Web Page

Listing 7-2 is the HTML source code for Figure 7-1's Web page when it has been served by a TINI using servlets. The Web page's HTML code isn't stored in a separate file. Instead, the servlet generates the page on request.

The HTML code is much the same as in Listing 7-1's code for the Rabbit, with differences only in how the dynamic data is handled.

```
<html>
<head>
  <title>Device Controller </title>
</head>

<body>
<h1> Device Controller Demo</h1>

<table>
<tr>
  <td><img src ="ledon.gif" ></td>
  <td><img src ="ledoff.gif" ></td>
</tr>

<tr>
  <td>
  <a href="/servlet/DeviceController?button1">
    <img src="button.gif"></a>
  </td>
  <td>
  <a href="/servlet/DeviceController?button2">
    <img src="button.gif" ></a>
  </td>
</tr>

</table>

<p>LED 1 is on.</p>
<p>LED 2 is off.</p>
<p>Click a button to turn an LED on or off.</p>
<p>The Web page will update to show the current states of the
 LEDs.</p>

</body>

</html>
```

Listing 7-2: When a browser requests Figure 7-1's Web page from a TINI module running the Tynamo Web server and the DeviceController servlet code in this chapter, the server inserts the file names and text descriptions to match the LEDs' states

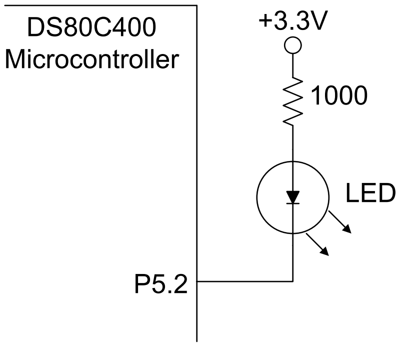


Figure 7-2: A port bit on the DSTINIm400 module controls an LED. A logic low turns the LED on.

The hyperlinks for the buttons each contain a string that names the servlet being requested. This Java statement sends the text required to place the image `button.gif` on a page and make it a hyperlink:

```
out.print("<p>
  <a href=\"/servlet/DeviceController?button1\">
  <img src=\"button.gif\">
  </a>"</p>);
```

In the hyperlink, `/servlet/DeviceController` matches a mapping in a configuration file for the Web server. The mapping tells the server to run the `DeviceController` servlet. Following the servlet mapping is a question mark and a query string (`button1`) that identifies the button that was clicked.

The server inserts the `img` tags (`"ledon.gif"`, `"ledoff.gif"`) and the text descriptions of the LEDs (`LED 1 is on.`, `LED 2 is off.`) in the page each time the page is served. The images and text match the current states of the LEDs.

LED1 is D1 on the DSTINIm400 module and is controlled by Port 5, bit 2 on the DS80C400 microcontroller. Figure 7-2 shows the interface. A logic low on the port bit sinks current to turn the LED on, and a logic high cuts off the current and turns the LED off. The 1-kilohm pull-up resistor limits current through the LED. LED2 is an optional LED that can interface in the same way to Port 5, bit 3 on the '80C400.

(The DS-TINI-1 module includes an LED controlled by Port 3, bit 5 on the DS80C390 microcontroller. This interface is the reverse of the '80C400's circuit: a logic high turns the LED on, and a logic low turns it off.)

The Servlet

The Device Controller servlet is a Java class with methods that serve Figure 7-1's page and respond when users click the buttons on the page.

The class imports `java.io` classes to enable reading inputs and writing to outputs. Two additional packages support servlets. The `javax.servlet` package includes the `Servlet` interface and the abstract class `GenericServlet`, which implements the `Servlet` interface. The `javax.servlet.http` package contains the abstract class `HttpServlet`, which adds support for HTTP and Web applications. The TINI-specific class `com.dalsemi.system.BitPort` enables accessing the port bits that control the TINI's LEDs.

The `DeviceController` class extends the `HttpServlet` class of the `javax.servlet.http` package.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.dalsemi.system.BitPort;

public class DeviceController
  extends HttpServlet
{
```

Two `BitPort` objects, `led1` and `led2`, correspond to port bits on the '80C400 microcontroller The `BitPort` class's `readLatch` method returns the last value written to a port bit, which in turn indicates the state of the LED controlled by the bit. (To read a port bit configured as an input, use the `read()` method.)

```
    BitPort led1 = new BitPort(BitPort.Port5Bit2);
    BitPort led2 = new BitPort(BitPort.Port5Bit3);
```

Performing Tasks on Startup

The `GenericServlet` class includes an `init()` method that enables a servlet to perform tasks on startup. The `init()` method is called once, when the servlet starts, and is optional. In this example, `init()` sets the LEDs' port bits to turn the LEDs off.

```
public void init() throws ServletException {
  led1.set();
  led2.set();
} // end init()
```

Serving GET Requests

A received HTTP GET request causes the `DeviceController` class's `doGet()` method to be called. The `DeviceController` class overrides `HttpServlet`'s `doGet()` method with a method that serves the Device Controller Web page to the client. The `doGet()` method has two parameters: `request` is an `HttpServletRequest` object that contains the client's request, and `response` is an `HttpServletResponse` object that contains response information for the client.

A `ServletException` occurs if the server can't handle the GET request for some reason. An `IOException` occurs if there is an input or output error when the servlet is handling the `GET` request. The `doGet()` method throws `ServletExceptions` and `IOExceptions`.

```
public void doGet
  (HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
```

The `getQueryString()` method of the `HttpServletRequest` object returns a string received from the client in the GET request. In this application, the query string is the text `button1` or `button2` that appears after the question mark in a button's hyperlink.

```
  String query = request.getQueryString();
```

If the query string equals `"button1"`, indicating that Button 1 was clicked, the program calls the `toggle()` method to change the state of `led1`. The method returns `true` if the toggled LED is on and `false` if the LED is off.

If the query string doesn't contain `"button1"`, the program uses the `BitPort` class's `readLatch()` method to find out the last value written to `led1`'s port bit. If the last value written was zero, the LED is on and `ledOn` is set to `true`. If the last value written was 1, the LED is off and `ledOn` is set to `false`.

In the same way, depending on the contents of the query string, the program toggles or just reads the state of `led2`.

```
boolean led1On;
if ("button1".equals(query)) {
  System.out.println("Button 1 was clicked");
  led1On = toggle(led1);
} else {
  led1On = (led1.readLatch() == 0);
}

boolean led2On;
if ("button2".equals(query)) {
  System.out.println("Button 2 was clicked");
  led2On = toggle(led2);
} else {
  led2On = (led2.readLatch() == 0);
}
```

If there is no query string, such as when a user requests the page for the first time, the code reads the states of both LEDs and toggles neither.

Two `String` variables (`led1Image, led2Image`) hold the names of image files that correspond to the LEDs' states. Two additional `String` variables (`led1State, led2State`) hold the text `"on"` or `"off"` as appropriate for the LEDs. After reading the LEDs' states, the program sets the image and text strings to match the LEDs.

```
String led1Image;
String led1State;
if (led1On) {
  led1Image= "/ledon.gif";
  led1State = "on";
}
else {
  led1Image = "/ledoff.gif";
  led1State = "off";
}
```

```
    String led2Image;
    String led2State;
    if (led2On) {
      led2Image= "/ledon.gif";
      led2State = "on";
    }
    else {
      led2Image = "/ledoff.gif";
      led2State = "off";
    }
```

A call to the class's `sendWebPage()` method sends an updated Web page to the client. The method uses the `HttpServlet` response object and the four variables that indicate the LEDs' states and image files.

```
    sendWebPage (response, led1Image, led2Image,
        led1State, led2State);
  } // end doGet()
```

Toggling an LED

The `toggle()` method toggles the state of a `BitPort` object. The method returns a boolean value that indicates if the corresponding LED is on. The value is `false` if the last value written to the LED was 1 and the LED is off, and `true` if the last value written was zero and the LED is on.

```
  private static boolean toggle(BitPort bitPort) {
    if (bitPort.readLatch() == 0) {
      bitPort.set();
      return false;
    } else {
      bitPort.clear();
      return true;
    }
  } // end toggle()
```

Sending the Web Page

The `sendWebPage()` method writes a Web page to an output stream. The method uses the `HttpServlet` response object and four variables that the method inserts in the Web page. The method throws `IOExceptions`.

```
  private void sendWebPage (HttpServletResponse response,
                            String led1Image,
                            String led2Image,
                            String led1State,
                                      String led2State)
        throws IOException {
```

The `setContentType()` method of the `HttpServletResponse` object sets the `Content-Type` field in the response's HTML header:

```
response.setContentType("text/html");
```

The `getOutputStream()` method of the `HttpServletResponse` object returns an instance of a `ServletOutputStream` object. The `ServletOutputStream` class extends the `java.io.OutputStream` class and provides an output stream for sending data to a client. You could use a `PrintWriter` object instead, but a `ServletOutputStream` object requires less processing, and thus is quicker. A series of `out.print` statements write the Web page's contents to the output stream. The servlet container automatically creates and writes an HTTP header that precedes the page's contents.

```
ServletOutputStream out = response.getOutputStream();
out.print("<html>"
    + "<head>"
    + "<title>Device Controller</title>"
    + "</head>"
    + "<body>"
    + "<h1>Device Controller</h1>");
```

Much of the HTML code is similar to the source code for the Web page in the Rabbit Device Controller example. The differences are in how the server gets the values of real-time variables and in how the server responds to button clicks.

The LED and button images are in a table to ensure they line up on the page. The variables `led1Image` and `led2Image` each contain a filename, `"ledon.gif"` or `"ledoff.gif"`, as appropriate, to indicate the images the browser should display for the LEDs.

Some of the text in the HTML code includes quotation marks. Because quotation marks are also the delimiters for a string, any quotation mark within a string must be preceded by a back slash (\). For example, many HTML tags include attributes enclosed by quotation marks, such as `<img src="ledon.gif">`

To write this line to the output stream, each quotation mark in the string must be preceded by a back slash: `out.print("<img src=\"ledon.gif\">");`

The back slash indicates that the quotation mark is part of the string and not the string's delimiter.

```
out.print("<table>"
    + "<tr>"
    + "<td>"
    + "<img src=\"");
out.print(led1Image);
out.print("\">"
    + "</td>"
    + "<td>"
    + "<img src=\"");
out.print(led2Image);
out.print("\">"
    + "</td>"
    + "</tr>");
```

In the hyperlinks for the button images, `/servlet/DeviceController` is a mapping that tells the server, via a configuration file, that `DeviceController` is a servlet. When a user clicks a button on the Web page, the browser returns either `button1` or `button2` to the TINI in the request's query string.

```
out.print("<tr><td>");
out.print("<a href=
    \"/servlet/DeviceController?button1\">
    <img src=\"/button.gif\"></a>");
out.print("</td><td>");
out.print("<a href=
    \"/servlet/DeviceController?button2\">
    <img src=\"/button.gif\" ></a>");
out.print("</td></tr></table>");
```

In addition to the LEDs' images, two lines of text indicate the states of the LEDs. The variables `led1State` and `led2State` each hold the text `"on"` or `"off"` as appropriate, and `out.print` statements write the text to the output stream.

```
out.print("<p>LED 1 is ");
out.print (led1State);
out.print(".</p>"
                + "<p>"
                + "LED 2 is ");
            out.print (led2State);
            out.print(".</p>"
                + "<p>"
                + "Click a button to turn an LED on or off."
                + "<p>"
                + "The Web page will update to show the
                    current states of the LEDs."
                + "</p>"
                + "</body>"
                + "</html>");
```

The servlet container flushes and closes the output stream when the request has been serviced, so there's no need to do so in the servlet.

Loading and Running Servlets

In writing an ordinary Java program for use on a TINI, you compile the program to one or more *.class* files and use the TINIConvertor utility to convert the file(s) to a *.tini* file. You can then use an FTP program to copy the file to the TINI. Or you can use the build utility Ant to automate the process of creating and copying the files. When the file or files have been transferred to the TINI, you can run the program from a Telnet session by typing `java`, followed by the name of the *.tini* file.

With servlets, things are more complicated. The Tynamo Web server functions both as a Web server, which responds to HTTP requests, and as a servlet container, which contains and manages the servlets. A variety of configuration files contain information about the servlets and Web server. The Ant utility is the recommended way to compile, convert, and deploy the Web server and servlets on the TINI.

With Ant, you can compile your *.java* files and create the file *webserver.tini*, which contains both the object code required to respond to HTTP requests and the code for your servlets.

If you use another servlet container, such as TiniHttpServer, the details will vary, but the information about how to use Ant, TiniAnt, and the configuration files are likely to be similar.

Required Components

These are the required components for creating and running servlets on a TINI with the Tynamo Web server:

- A TINI module, to run the Tynamo Web server.
- The Java SDK, for program development, from *java.sun.com*.
- The Tynamo Web server, to support servlets, from *tynamo.qindesign.com*.
- Ant, a Java-based build tool, from *jakarta.apache.org*.
- TiniAnt, a plug-in that integrates TINI's build process into Ant, from *tiniant.sourceforge.net*.
- The *NetComponents.jar* library, with FTP and Telnet support for deploying the Web server on the TINI. The NetComponents distribution is available from *www.savarese.org*.

The Tynamo Web server uses four configuration files. You must edit at least three of these to provide information that is specific to your development PC and your servlets. The *build.properties* file contains the locations and names of various files and directories on the development computer. The *servlets.props* file contains information about the servlets. The *deploy.properties* file contains your TINI's IP address and other information that the Ant utility uses in copying the Web server's files to the TINI. The *webserver.props* file enables you to specify a default directory, home page, and other properties of your server. (Many servers can use the default *webserver.props* file, with no editing.) You can edit these files in any text editor.

Below is more information about each of these files, followed by instructions for how to use the files in compiling, converting, and deploying files to the Web server.

Creating a build.properties File

The *build.properties* file is in the home directory of the Tynamo distribution. The file contains the locations and names of the TINI's home directory and the servlets on the development computer. The Ant utility uses the information in the file in building the TINI's executable file. Listing 7-3 is an example *build.properties* file.

```
#example build.properties file
tini.path=C:/tini1.11
src.paths=/myservlets
src.files=DeviceController.java, FormResponse.java
include.servletReloading=false
dependency.files=
dependency.groups=
dependency.classpath=
reflect.classes=DeviceController, FormResponse
```

Listing 7-3: The build.properties file contains information that Ant uses in compiling the servlets.

For each of the following items, edit the existing text by inserting the information that applies to your system and servlets. Use forward slashes as separators even if the operating system of your developement computer (such as Windows) uses back slashes. Ant converts to back slashes as needed.

Set `tini.path` equal to the location of the TINI SDK on the development computer:

```
tini.path=/tini1.11
```

Set `src.paths` equal to the location of the source code for your servlets on the development computer:

```
src.paths=/myservlets
```

If there are multiple locations, separate the paths with colons or semicolons:

```
src.paths=/myservlets;/test
```

Set `src.files` equal to the names of your servlets, separating multiple names with commas or spaces:

```
src.files=DeviceController.java, FormResponse.java
```

Set `reflect.classes` equal to the full class name of each servlet, separating multiple names with commas or spaces:

```
reflect.classes=DeviceController, FormResponse
```

Three dependency entries can contain information about the classes that a servlet uses, or depends on. Not every servlet requires dependency information.

A `dependency.files` entry specifies the name and location of a file that contains dependency information for one or more servlets. An example entry is:

```
dependency.files=examples/servlet_examples_dep.txt
```

Below is the information provided in the dependency file for Tynamo's example servlet `RequestInfoServlet`:

```
RequestInfoServlet=
com.qindesign.servlet.example.RequestInfoServlet;
com.qindesign.servlet.example.Common
```

The `RequestInfoServlet` entry has two values separated by a semicolon. The first value is the full name of the servlet's class. The second value informs the build process that the servlet depends on the `com.qindesign.servlet.example.Common` class.

Creating a servlets.props File

The *servlets.props* file is in the *\bin* directory of the Tynamo installation and must contain information required by the servlet container to run your servlets. The file provides information about each servlet supported by the server. Listing 7-4 is an example *servlets.props* file for the servlets `DeviceController` and `FormResponse`.

A servlet name identifies the servlet in the file. The servlet names in the example are `DeviceController` and `FormResponse`. A mapping specifies how clients can request to run the servlet and has the following format:

*servlet_name***.mapping=***mapping*

where *servlet_name* is a servlet name and *mapping* is the text that clients can use to request the servlet from the server.

The following mapping enables clients to request to run the servlet `DeviceController` by typing the TINI's IP address or domain name followed by `/servlet/DeviceController` in a browser's Address text box:

```
DeviceController.mapping=/servlet/DeviceController
```

For example, if the IP address is 192.168.111.9, the user would enter the following:

```
DeviceController.mapping=/servlet/DeviceController
DeviceController.class=DeviceController

FormResponse.mapping=/servlet/FormResponse
FormResponse.class=FormResponse
```

Listing 7-4: The servlets.props file contains configuration information for your servlets.

```
http://192.168.111.9/servlet/DeviceController
```

The `servlets.props` file must also specify the full class name of the class that implements the `javax.servlet.Servlet` interface for each servlet. The class name is the name of the servlet's class in the source code, preceded by its package name, if any. This information uses the following format:

servlet_name.**class=***class*

where *servlet_name* is the servlet name and *class* is the class name. In the example, the class name for the servlet `DeviceController` is also `Device-Controller`. In this case it seems redundant, but other classes might use a different name for the class name and servlet name.

If the servlet is in a package, the class name must specify the package name as well, as in this example:

```
Shutdown.class=com.qindesign.servlet.ShutdownServlet
```

The optional `initParams` entry can specify one or more initialization parameters to use when the servlet starts:

```
Shutdown.initParams=passwd=shut:down
```

The optional `loadOnStartup` entry can specify that the servlet should load when the server starts, rather than on first use:

```
Shutdown.loadOnStartup=true
```

The number sign (`#`) indicates a comment, which the server ignores:

```
# Shutdown servlet
```

The example *servlets.props* file included with the Tynamo Web server has additional examples.

Creating a deploy.properties File

The *deploy.properties* file simplifies the process of transferring files to a TINI. Listing 7-5 shows an example.

Four `deploy` properties contain information about the TINI. The `server` property is the TINI's IP address. The `userid` and `password` properties are the user ID and password required to log onto the TINI's FTP server. The `rootdir` property is the directory the deploy process should use as the root directory on the TINI when transferring files. The deploy process creates the directory if it doesn't exist.

Setting Web Server Properties

The *webserver.props* file enables you to specify properties of the server. The default file will work with no changes, but you can edit the entries if you wish. To use a default directory other than */web/http-root* for files on the server, edit this entry with the desired directory path.

```
server.rootDir=/web/http-root
```

To use a default home page other than *index.html*, edit this entry with the desired default file's name:

```
server.welcomeFile=index.html
```

The entries in the provided file show additional options you can change.

Running the Web server

When you've obtained the necessary components and have written a servlet such as the DeviceController servlet above, these are the steps required to use Tynamo to run the servlet on a TINI:

1. Install Ant and TiniAnt on your development PC, following the instructions provided with each, including setting the recommended environment variables to identify file locations.

2. As described above, edit *build.properties, servlets.props, deploy.properties,* and *webserver.props* with the appropriate information for your TINI and servlets.

```
deploy.server=192.168.111.2
deploy.userid=root
deploy.password=tini
deploy.rootdir=/web
```

Listing 7-5: The deploy.properties file contains information specific to the TINI that will run the Web server.

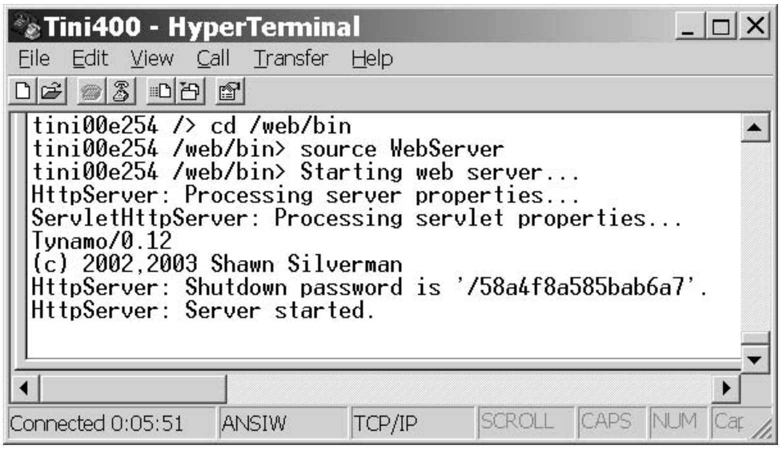
3. Build *webserver.tini* with Ant. Open a window with a command prompt. Under Windows XP, click **Start**, then **Run**, and enter **cmd** in the **Open:** text box that appears. Change to Tynamo's home directory and enter **ant**. This runs the file *ant.bat* included with the Ant distribution. Ant uses the information in *build.properties* and Tynamo's *build.xml* file to locate the needed files, compile, and convert the result to the file *webserver.tini*. The file contains the executable code for the servlet container and the servlets the Web server can run.

4. Copy any static HTML files, images, or other files the Web server will need to access to the appropriate directories under the Tynamo's home directory on the development computer. The *webserver.props* file specifies the root directory for these files. The default is *http-root*.

5. From a command prompt in Tynamo's home directory, enter `ant deploy`. This runs *ant.bat* again, but this time runs the `deploy` task instead of the default `build` task. (Tynamo's *build.xml* file specifies the default task.)

The deploy task copies the Web server's files to the TINI. Using the default settings, the files copied are the following files under Tynamo's home directory:

\bin\webserver.tini (the Web server application)
\bin\WebServer (a script to run the Web server)
\bin\webserver.props (configuration information about the Web server)
\bin\servlets.props (information about the servlets)
\bin\mimeTypes.props (MIME definitions for file types)
\http-root\* (all files in this directory)


Tini400 - HyperTerminal
File Edit View Call Transfer Help
Connected 0:05:51 ANSIW TCP/IP SCROLL CAPS NUM Cap

```
tini00e254 /> cd /web/bin
tini00e254 /web/bin> source WebServer
tini00e254 /web/bin> Starting web server...
HttpServer: Processing server properties...
ServletHttpServer: Processing servlet properties...
Tynamo/0.12
(c) 2002,2003 Shawn Silverman
HttpServer: Shutdown password is '/58a4f8a585bab6a7'.
HttpServer: Server started.
```

Figure 7-3: You can run the Tynamo server from a Telnet session with the TINI.

6. To run the Web server, in a Telnet session, at a command prompt in the root directory, enter the following command:

```
source web/bin/WebServer
```

This executes the WebServer script, which contains the following text:

```
java /web/bin/webserver.tini /web/bin/webserver.props &
```

On running the Web server, the Telnet window displays something like the text in Figure 7-3. And the TINI is ready to run the servlets named in `servlets.props`.

Serving Other Files

The default configuration of the Tynamo Web server treats any request not handled by a servlet as a request for a file under the */web/http-root* directory. Examples of such requests include requests for image files or static HTML files. If the request doesn't specify a valid servlet or file name, the default configuration serves the page *index.html* if available in the specified directory.

The default home page can contain a hyperlink to the Device Controller servlet:

```
<a href="/servlet/DeviceController">Device Controller</a>
```

Save the Web page as *index.html* and copy the file to the TINI's */web/http-root* directory. Then users who enter the TINI's IP address alone or the IP address followed by */index.html* will see the Web page with the link to the servlet.

To redirect the user's browser to request the servlet automatically from the home page, include this META tag in the Web page's HEAD section:

```
<meta http-equiv="Refresh" content="0;
  url=/servlet/DeviceController">
```

In Depth: Using CGI and Servlets

The examples above showed how Web pages can use CGI functions and servlets to enable users to click hyperlinks to run program code on the server and view the result in a Web page. This section has more detail about CGI and servlets, including additional examples that show how embedded systems can use forms to accept text input from users.

CGI for Embedded Systems

The common gateway interface (CGI) defines a protocol that enables users to click a link or button on a Web page to request a server to execute program code. A CGI program can perform just about any function on the server. After running the requested program, the server returns a result in an HTTP response.

Support for CGI programming has been around since the earliest days of the Web. The first Web server to implement CGI was the NCSA HTTPD server from the National Center for Supercomputing Applications. NCSA publishes a CGI Specification at *http://hoohoo.ncsa.uiuc.edu/cgi/*. Many embedded systems that support networking also include support for CGI. The Dynamic C library *http.lib* is an example.

CGI programming doesn't require a particular programming language. On large servers, the Perl language has long been popular. Perl programs are typically scripts that require an interpreter to execute, and large servers generally have a Perl interpreter. A small embedded system isn't likely to have a Perl interpreter, so CGI programs for embedded systems are often written in C.

A Web server that runs CGI programs must be able to do the following:

- Identify a received HTTP request that references a CGI program to execute.
- Locate and run the requested CGI program.
- Return an HTTP response.

The response the server returns after running a CGI program often includes an HTTP redirection code that advises the browser to request a page containing an acknowledgment or a refreshed page with updated data.

Some CGI programs process data submitted by a client on a form. When a client submits a request that contains form data, the server must be able to pass the data to the CGI program that will use the data.

For security reasons, a server may provide a way to enable, disable, or limit support for CGI.

CGI Requests

A client can request a server to run a CGI program by sending an HTTP request containing the name of a CGI program on the server. In the Device Controller example in this chapter, the buttons on the Web page are hyperlinks that each contain a program name:

```
<a href="/led1toggle.cgi"> <img src="button.gif"> </a>
```

Clicking the image of the button causes the browser to request the server to run the program (or function) *led1toggle.cgi*.

Text hyperlinks are another way to request a server to run CGI programs. The following HTML code causes the text "Turn off LED1" to appear on a Web page as a hyperlink:

```
<a href="/led1off.cgi"> Turn off LED1> </a>
```

Clicking the hyperlink causes the browser to request the server to run the program *led1off.cgi*.

Servers also use CGI programming in accepting input from a Web page containing a form. Clicking a form's **Submit** button causes the form's data to be sent to the server in an HTTP GET or POST request. The server can be configured to respond to the request by running CGI code that processes the form data and returns a response.

Identifying and Running CGI Programs

CGI code may be an interpreted script, a compiled program, or a function within a program.

Large servers often store all CGI programs in a directory such as *cgi-bin*. Or a server may identify CGI programs by a *.cgi* extension in the program name. In Dynamic C, CGI programs can be functions declared as `HTTPSPEC_FUNCTION` items in an `HttpSpec` structure. Or an application can use the form-handling capabilities in Dynamic C's server utility library (*zserver.lib*) to process form data.

Returning a Response

A CGI program must return an HTTP response to the request that caused the server to run the program. Like other HTTP responses, the response includes a status line, response headers, and if appropriate, a message body. The response can provide requested information or acknowledge that submitted data was received. To enable a user to view the result of executing a CGI program, a response may contain a redirection code that advises the user's browser to refresh the current Web page.

In this chapter's Device Controller application for the Rabbit module, after a user clicks a button on the Web page, the browser requests a refreshed copy of the page so the user can see the LEDs' current states. To cause the browser to request to refresh the page, the server returns a response containing the following code in the response line, with the desired file name and path in a `Location` header:

```
Http 1.0 302 Found
Location: http://192.168.111.7/index.shtml
```

On receiving this response code in reply to a `GET` request, the browser sends a new `GET` request for the specified file. In case a browser doesn't support automatic redirection, many responses include a message body that displays a hyperlink to the file in the `Location` field and text that advises the user to click the link to view the file.

Servlets for Embedded Systems

For displaying real-time data and responding to user input, Java programmers can use servlets, as introduced in the TINI example in this chapter. A servlet is a Java class that adds capabilities to a server.

A Web server that runs servlet code must have a container, or servlet engine, to manage the servlets. The container provides network services for sending and receiving requests, decodes requests, and formats responses. For security, a container can also place restrictions on the execution of servlets.

A browser that requests a Web page served by a servlet doesn't require support for Java or servlets. When a browser sends a URL to a server, the browser doesn't have to know or care whether the URL identifies a static Web page or a servlet. The text that the browser sends to the server identifies the servlet. In the DeviceController example, the images of buttons are hyperlinks that users can click to request the server to run the servlet DeviceController:

```
<a href="/servlet/DeviceController?button1">
  <img src="button.gif"></a>
```

A mapping in the server's configuration file identifies `/servlet/DeviceController` as a servlet, and `button1` following the question mark is the query string that the browser returns to the server along with the requested URL.

On receiving a request for a servlet, the server runs the servlet code. The servlet can generate dynamic data, insert the data into a Web page, and write the Web page's contents to an output stream for sending to the client. A servlet can also do just about anything an ordinary Java program can do, such as making calculations, performing logical operations, and reading and writing to files or ports.

The document that defines servlets and their behavior is the Java Servlet Specification, available from *java.sun.com*.

On receiving an HTTP request containing the name of a servlet to run, the server passes the request to the servlet container. The container examines the request to determine which servlet to call. The container then calls the servlet, passing two objects: a request object with information about the request and a response object that will contain information about the response. A response object may supply an `OutputStream` or `PrintWriter` object that the servlet uses to respond to the request. The servlet runs, performing its programmed function and returning a response to the request.

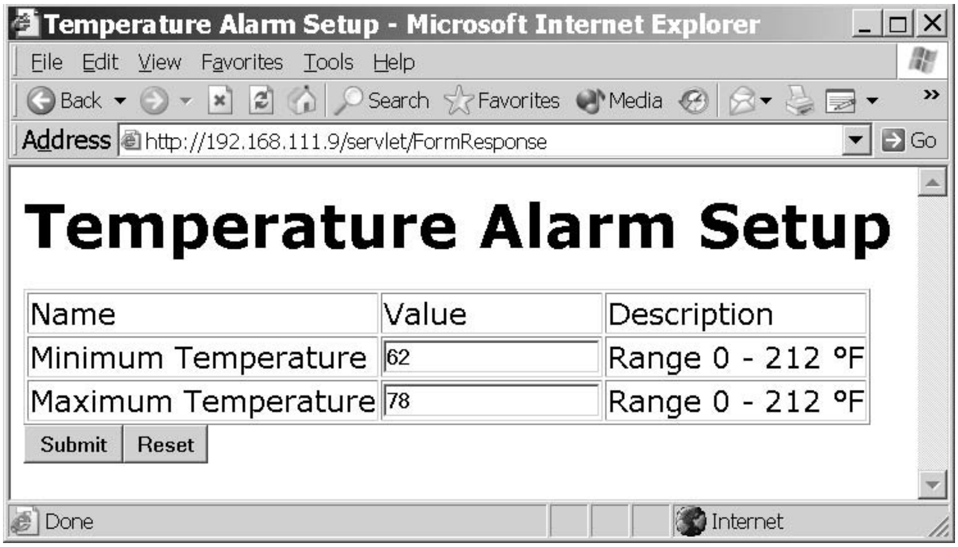
The `HTTPServer` class in the TINI's `com.dalsemi.tininet.http` package supports static Web pages only. The Tynamo Web server and TiniHttpServer are more powerful alternatives that add support for servlets.

Receiving Form Data

In addition to providing information on Web pages, servers can receive information from users by hosting Web pages that contain forms. A form can contain text boxes or other elements where users can enter data or make selections. When the user clicks a form's **Submit** button, the browser sends the form data to the server in an HTTP GET or POST request. The server can use the data in any way. An embedded system might use a form to request configuration data, collect information about users, or request passwords. The server may return a page that acknowledges receiving the data or a response that redirects the user's browser to another page.

As Chapter 6 showed, the HTML standard includes tags and attributes for creating forms on a Web page. The examples below show how to host forms on Rabbit and TINI modules.

Figure 7-4 shows a form that enables the user to enter maximum and minimum temperatures for use in an alarm system. When a user clicks **Submit**, the browser sends the temperature values to the server in an HTTP request. On receiving the values, the server either returns the Web page in Figure 7-5 or a response that instructs the user's browser to request the page.


Temperature Alarm Setup - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.9/servlet/FormResponse
Go
Done
Internet

Temperature Alarm Setup

Name	Value	Description
Minimum Temperature	62	Range 0 - 212 °F
Maximum Temperature	78	Range 0 - 212 °F

Submit Reset

Figure 7-4: This Web page contains a form that enables users to enter values for use by the server.

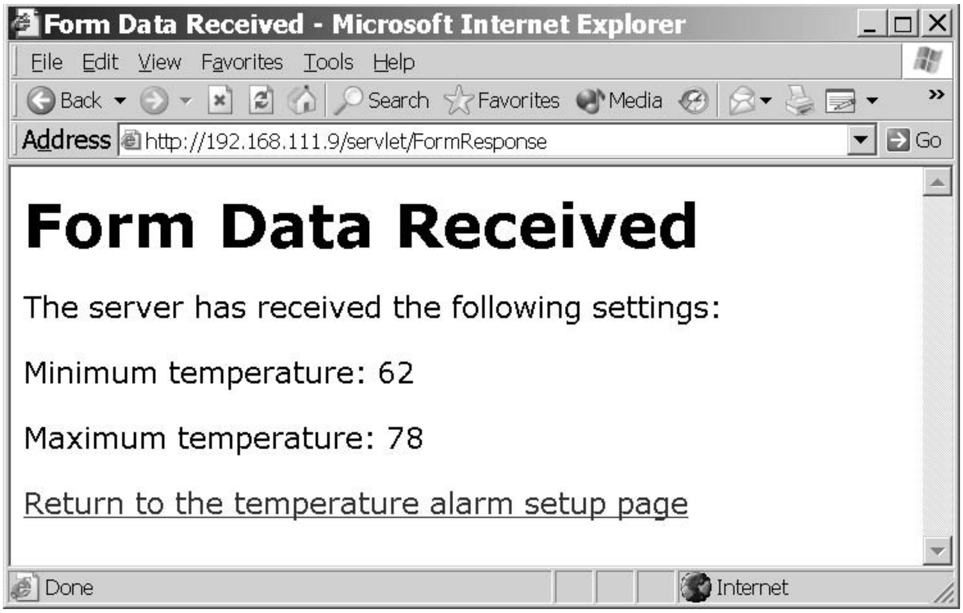
A server could use the temperature values to configure a temperature alarm system. In a similar way, you can use forms in just about any application where the server wants to collect information via a Web page.

Listing 7-6 is the HTML code for the form. Every form has three elements: `form` tags that define the start and end of the form, one or more controls that enable users to provide data to the server, and a **Submit** button that enables users to send the data to the server. In addition, most forms include descriptive text and a **Reset** button that returns the inputs to the values they contained when the page was served (before the user made any changes).

In Listing 7-6, the opening tag of the form is:

```
<form action="/" method="post">
```

The `FORM` tag's `action` attribute names the URL where the browser will submit the form data when a user clicks the **Submit** button. In this example, the URL is `"/"`, which refers to the server's default file. The `method` attribute's `"post"` value may be lower case in the HTML file. When the


Form Data Received - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.9/servlet/FormResponse
Go
Done
Internet

Form Data Received

The server has received the following settings:

Minimum temperature: 62

Maximum temperature: 78

Return to the temperature alarm setup page

Figure 7-5: A server might return a page like this to acknowledge receiving form data from a user.

browser sends a request, POST is upper case as required by the HTTP standard.

The `method` attribute specifies whether the browser will use an HTML GET or POST request to send form data to the server. In a GET request, the browser appends the data to the URL being requested. In a POST request, the browser places the data in the body of the request.

The form's closing tag is `</form>`. Everything between the form's opening and closing tags is part of the form.

Listing 7-6's form uses an HTML table to format the information in the form. Each variable has a name, value, and description in the table. The names and descriptions are plain text, except for the degree symbol. The HTML code `&deg;` causes the browser to display a degree symbol.

The `input` tags determine the contents of the cells in the Value column. This example `input` tag has four attributes:

```
<input type="text" name="maximum_temperature"
```

```
<html>
<head><title>Temperature Alarm Setup</title></head>

<body>
<h1>Temperature Alarm Setup</h1>

<form action="/" method="POST">

<table border>

<tr>
<td>Name</td>
<td>Value</td>
<td>Description</td>
</tr>

<tr>
<td>Minimum Temperature</td>
<td><input type="text" name="minimum_temperature"
    value="72" maxlength="3"></td>
<td>Range 0 - 212 &deg;F</td>
</tr>

<tr>
<td>Maximum Temperature</td>
<td><input type="text" name="maximum_temperature"
    value="78" maxlength="3"></td>
<td>Range 0 - 212 &deg;F</td>
</tr>

</table>

<p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</p>

</form>
</body>
</html>
```

Listing 7-6: HTML source code for Figure 7-4's form.

```
    maxlength="3" value="80">
```

The value of the `type` attribute is set to `"text"` to specify that the input is a single-line text box or other input control for entering text. The `name` attribute identifies the control on the form. The `maxlength` attribute is the maximum number of characters a user may enter in the text box. The `value` attribute is the default data the text box displays.

In addition to text, a variety of other controls use `input` tags, including check boxes, radio buttons, passwords, **Submit** buttons, and **Reset** buttons. Every `input` tag must have a `value` attribute. All types except text (the default) must have a `type` attribute, and most tags require a `name` attribute. The other attributes needed vary with the input type and the application's requirements.

Two additional `input` tags in Listing 7-6 add **Submit** and **Reset** buttons to the form. For each, the `type` attribute specifies the button type and a `value` attribute specifies the text to display on the button. When a form has just one **Submit** button and one **Reset** button, the `type` attribute identifies the buttons and there's no need for `name` attributes to further identify the buttons.

Forms on a Rabbit

The RCM3200 RabbitCore module can host Figure 7-4's form. In the Device Controller example earlier in this chapter, an `#ximport` directive loads a file containing the Web page's HTML source code into the RabbitCore's memory. For forms, instead of providing an HTML file containing the form to serve, you can use functions in Dynamic C's server utility library, *zserver.lib,* to create a form from information provided in the application.

An advantage of using *zserver.lib* to create forms is its automatic handling of errors in user input according to limits you specify. A limitation of using *zserver.lib* is the need to use its 3-column table format, unless you modify the library's display handler.

The following code shows how an RCM3200 module can serve the Temperature Alarm form and response.

Initial Defines and Declarations

Again, much of the configuration code is similar to the code in previous examples. `TCPCONFIG` specifies a macro that sets a network configuration stored in the file *tcp_config.lib*. Your program must specify an appropriate macro for your system and network configuration, as described in Chapter 5.

```
#define TCPCONFIG 1
```

Because of the need for forms support, this program uses the `ServerSpec` structure defined in *zserver.lib*, which includes support for basic forms, instead of the `HttpSpec` structure in *http.lib*. When `HttpSpec` is unneeded, the `HTTP_NO_FLASHSPEC` directive saves code space.

```
#define HTTP_NO_FLASHSPEC
```

The `FORM_ERROR_BUF` directive is required for forms. The directive reserves memory for a buffer used in form processing and must be large enough to hold the name, value, and four additional bytes for each form variable.

```
#define FORM_ERROR_BUF 256
```

On receiving form data, the server redirects the client's browser to the URL specified in `FORM_RESPONSE_REDIRECTTO`. In this application, the URL points to a file that acknowledges receiving the form data. `REDIRECTHOST` is the `_PRIMARY_STATIC_IP` address defined in *tcp_config.lib*.

```
#define REDIRECTHOST _PRIMARY_STATIC_IP
#define FORM_RESPONSE_REDIRECTTO
  "http://" REDIRECTHOST "/formresponse.shtml"
```

All C functions not declared as root go to extended memory. The *dcrtcp.lib* library supports IP and TCP. The *http.lib* library supports HTTP functions.

```
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
```

The `#ximport` directive loads a file from the development PC into the Rabbit's Flash memory. The directive associates the symbol `form_response_shtml` with the file's address in memory.

```
#ximport "c:/rabbitserver/formresponse.shtml"
    form_response_shtml
```

The `HttpType` structure specifies the handler to use with different file extensions. Web pages that contain SSI directives have the extension *.shtml* and use Dynamic C's SHTML handler. Plain HTML pages use the default HTML handler.

```
const HttpType http_types[] =
{
   { ".html", "text/html", NULL},
   { ".shtml", "text/html", shtml_handler}
};
```

Responding to Submitted Data

On receiving form data, the `form_response()` function executes and calls the `cgi_redirectto()` function. This function causes the server to return an HTTP response that redirects the client's browser to the Web page named in `FORM_RESPONSE_REDIRECTTO`.

```
int form_response(HttpState* state)
{
  cgi_redirectto(state, FORM_RESPONSE_REDIRECTTO );
  return 0;
} // end form_response()
```

The main() Function

The program's `main()` function creates the form, initializes the TCP/IP stack and Web server, and enters an endless loop that processes received HTTP requests and can perform any other tasks the system is responsible for.

The *zserver.lib* library includes a `ServerSpec` structure that has information about the files, functions, and variables that the server can access. The library contains functions that access elements in the structure. One of the items in the `ServerSpec` structure is an array of `FormVar` structures that hold information about a form's variables. This application has two form variables, so it defines an array (`setup`) that contains two `FormVar` structures. The `form`, `function`, and `var` variables are values returned by `ServerSpec` functions.

```
void main(void) {
  FormVar setup[2];
```

```
    int form;
    int function;
    int var;
```

The `maximum_temperature` and `minimum_temperature` variables are the form variables that users can change.

```
    int maximum_temperature;
    int minimum_temperature;
    maximum_temperature = 212;
    minimum_temperature = 0;
```

Creating the Form

A series of `ServerSpec` functions sets up the form and configures the server to serve the form and the page sent in response to receiving form data.

Adding a Web page. The `sspec_addxmemfile()` function names the Web page that users will see after submitting form data:

```
  sspec_addxmemfile ("formresponse.shtml",
    form_response_shtml, SERVER_HTTP);
```

The function has three parameters:

`"formresponse.shtml"` is the name of the file containing the Web page on the server.

`form_response_shtml` is the location where the `#ximport` directive stored the file.

`SERVER_HTTP` indicates that the file is valid for Dynamic C's HTTP server. (Dynamic C also supports `SERVER_FTP`).

The function returns the location of the file in the `ServerSpec` structure or -1 on failure.

Adding a Form. The `sspec_addform()` function adds a form to the `ServerSpec` structure:

```
    form =
      sspec_addform ("setup.html", setup, 2, SERVER_HTTP);
```

The function has four parameters:

`"setup.html"` is the name of the form's Web page on the server.

`setup` is the `FormVar` array defined earlier.

`2` is the number of entries in the `setup` array.

`SERVER_HTTP` indicates that the form is valid for Dynamic C's HTTP server.

The value returned, `form`, is the form's location in the `ServerSpec` array.

The `sspec_setformtitle()` function sets the title the form will display:

```
sspec_setformtitle(form, "Temperature Alarm Setup");
```

The function has two parameters:

`form` is the value returned by `sspec_addform()`.

`"Temperature Alarm Setup"` is the title.

The function returns zero on success or -1 on failure.

Adding a Function. The `sspec_addfunction()` function adds a function to the list of objects the Web server recognizes.

```
function = sspec_addfunction("form_response",
 form_response, SERVER_HTTP);
```

The function has three parameters:

`"form_response"` is the function's name.

`form_response` is a pointer to the function.

`SERVER_HTTP` indicates that the function is valid for Dynamic C's HTTP server.

The `function` value returned is the function's location in the `ServerSpec` structure or -1 on failure.

Adding a Function to Call on Receiving Form Data. The `sspec_setformepilog()` function names the function that the server will call after receiving form data from a client:

```
sspec_setformepilog(form, function);
```

The function has two parameters:

`form` is the value returned by `sspec_addform()`.

`function` is the value returned by `sspec_addfunction()`.

The function returns zero on success or -1 on failure.

Specifying Form Variables

Another series of ServerSpec functions adds variables to the form and sets a name, description, number of characters, and range for each.

Adding a Variable. The `sspec_addvariable()` function adds a variable to the `FormVar` array in the `ServerSpec` structure. This is the function call for the first variable:

```
var = sspec_addvariable("maximum_temperature",
  &maximum_temperature, INT16, "%d", SERVER_HTTP);
```

The function has five parameters:

`"maximum_temperature"` is the variable's name on the form.

`&maximum_temperature` is a pointer to the variable.

`INT16` is the variable type.

`"%d"` specifies the output format on the form as a decimal number.

`SERVER_HTTP` indicates that the variable is valid for Dynamic C's HTTP server.

The value returned, `val`, is the function's location in the `ServerSpec` structure or -1 on failure.

The `sspec_addfv()` function adds a variable in a `FormVar` array to the form.

```
var = sspec_addfv(form, var);
```

The function has two parameters:

`form` is the value returned by `sspec_addform()`.

`var` is the value returned by `sspec_addvariable()`.

The value returned, `var`, is the index of the added form variable or -1 on failure.

Associating a Name with a Variable. The `sspec_setfvname()` function sets the name the form will display for the variable.

```
sspec_setfvname(form, var, "Maximum Temperature");
```

The function has three parameters:

`form` is the value returned by `sspec_addform()`.

`var` is the value returned by `sspec_addfv()`.

`"Maximum Temperature"` is the name to display on the form.

The function returns zero on success or -1 on failure.

Adding a Variable Description. The `sspec_setfvdesc()` function sets a variable description that the form will display:

```
sspec_setfvdesc(form, var, "Range 0 - 212 &deg;F");
```

The function has three parameters:

`form` is the value returned by `sspec_addform()`.

var is the value returned by `sspec_addfv()`.

`"Range 0 - 212 &deg;F"` is the text the form will display in the Description column for the `"Maximum Temperature"` variable.

The function returns zero on success or -1 on failure.

Setting a Variable's Maximum Length. The `sspec_setfvlen()` function sets the maximum number of characters the form will accept and display for a variable's value:

```
sspec_setfvlen(form, var, 3);
```

The function has three parameters:

`form` is the value returned by `sspec_addform()`.

`var` is the value returned by `sspec_addfv()`.

`3` is the maximum number of characters.

The function returns zero on success or -1 on failure.

Setting a Variable's Range. The `sspec_setfvrange()` function sets a variable's minimum and maximum allowed values.

```
sspec_setfvrange(form, var, 0, 212);
```

The function has four parameters:

`form` is the value returned by `sspec_addform()`.

var is the value returned by `sspec_addfv()`.

`0` is the minimum value the server will accept for the variable.

`212` is the maximum value the server will accept for the variable.

If a user enters a value outside the specified range, the server adds an error message to the form and redirects the user's browser to the form so the user can change the value. The function returns zero on success or -1 on failure.

Adding More Variables. In the same way, calls to these functions add the `minimum_temperature` value to the form:

```
var = sspec_addvariable("minimum_temperature",
  &minimum_temperature, INT16, "%d", SERVER_HTTP);
var = sspec_addfv(form, var);
sspec_setfvname(form, var, "Minimum Temperature");
sspec_setfvdesc(form, var, "Range 0 - 212 &deg;F");
sspec_setfvlen(form, var, 3);
sspec_setfvrange(form, var, 0, 212);
```

Accessing the Form

The `sspec_aliasspec()` function enables requesting the form in alternate ways. In this example, in addition to requesting the file *setup.html* by its file name, users can request the file `"index.html"` or the default Web page at the IP address (`"/"`).

```
sspec_aliasspec(form, "index.html");
sspec_aliasspec(form, "/");
```

Starting the Server

When the form has been created, the program is ready to initialize the TCP/IP stack and the Web server. As in the previous Rabbit HTTP example, calling `tcp_reserveport()` gives improved performance.

```
sock_init();
http_init();
tcp_reserveport(80);
```

The program's main loop calls `http_handler()` and can perform any other tasks the RCM3200 is responsible for. For example, for this application, the main loop might monitor temperatures and generate an alarm when a temperature is outside the minimum and maximum range specified on the form.

```
while (1) {
   http_handler();
}
```

```
} // end main()
```

Listing 7-7 is the HTML source code for the *formresponse.shtml* file in Figure 7-5. The page acknowledges receiving the form data and uses SSI `#echo` directives to display the temperature values received from the client.

When the RCM3200 module is running this program, users can access the form by entering the module's IP address or domain name in a browser's Address text box. Clicking the form's **Submit** button sends the temperature values to the RSM3200, which reads the values and either stores the values and returns an acknowledgment or returns an error message if either of the values is outside the accepted range.

Forms on a TINI

A TINI can serve Figure 7-4's form using the Tynamo Web server or another Web server with support for servlets. Listing 7-8 is the source code for the form when served by a TINI running a servlet. The only difference between the HTML code in Listing 7-6 and the form served by the TINI is the `form` tag's `action` attribute. For the TINI, form tag is:

```
 <form method=POST action="/servlet/FormResponse">
```

When a user clicks the **Submit** button, the browser submits the form data to the servlet `FormResponse` on the server. The server's configuration file identifies `/servlet/FormResponse` as a servlet.

Requesting the Servlet

When the Tynamo Web server and FormResponse servlet are loaded into a TINI, users can request the TINI to run the servlet by entering the TINI's IP address or domain name followed by `/servlet/` and the servlet's name:

```
 http://192.168.111.9/servlet/FormResponse
```

Or the TINI can contain a static Web page with a link to the servlet:

```
<A HREF="/servlet/FormResponse">View the Form</A>
```

The `FormResponse` servlet serves the form with the current values of `minimum_temperature` and `maximum_temperature` inserted. On receiv-

```
<html>
<head>
  <title>Form Data Received</title>
</head>

<body>
  <h1>Form Data Received</h1>

  <p> The server has received the following settings: </p>
  <p> Maximum temperature: <!--#echo
      var="maximum_temperature"--></p>
  <p> Minimum temperature: <!--#echo
      var="minimum_temperature"--></p>
  <P><a href="index.html">Return to the temperature alarm setup
      page</a></p>
</body>

</html>
```

Listing 7-7: HTML code for Figure 7-5‘s Web page when served by a Rabbit module. SSI directives retrieve the temperature values received when the client submitted the form.

ing new values from a client, the servlet returns a Web page that acknowledges receiving the values.

On receiving an HTTP GET request for the `FormResponse` servlet, the servlet returns a Web page that displays the current minimum and maximum temperature settings and enables users to change the values by typing new ones and clicking **Submit**. On receiving form data in a POST request, the servlet checks for valid data. If the submitted data is valid, the servlet returns a page that acknowledges receiving the data. If the data isn’t valid, the servlet returns the form with an error message and a request to retry.

The Servlet

As in the previous TINI example, to support servlets and HTTP, the program imports `javax.servlet` and `javax.servlet.http` classes for serv-

```
<html>
<head>
  <title>Temperature Alarm Setup</title>
</head>

<body>
  <h1>Temperature Alarm Setup</h1>

  <form method=POST action="/servlet/FormResponse">

  <table border>

  <tr>
    <td>Name</td>
    <td>Value</td>
    <td>Description</td>
  </tr>

  <tr>
    <td> Maximum Temperature </td>
    <td><input type="text" name="maximum_temperature"
        value=80></td>
    <td>Range 0 - 212 &deg;F</td>
  </tr>

  <tr>
    <td> Minimum Temperature </td>
    <td><input type="text" name="minimum_temperature"
      value=60></td>
    <td>Range 0 - 212 &deg;F</td>
  </tr>

  </table>

  <p></p>
  <p><input type="submit" value="Submit">
     <input type="reset" Value="Reset"></p>

  </form>
  </body>
  </html>
```

Listing 7-8: HTML source code for Figure 7-4's Web page using a servlet.

let support and `java.io` classes to support input and output functions. The `FormResponse` servlet extends the `HttpServlet` class.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FormResponse extends HttpServlet {
```

A `DELAY_TIME` constant determines how often the servlet executes a periodic task. The servlet uses the default values `DEFAULT_MIN_TEMPERATURE` and `DEFAULT_MAX_TEMPERATURE` if values previously set by the user aren't available. The *setup.bin* file stores the setup parameters from the `setupParameters` array. The timer thread enables the TINI to perform a task at timed intervals.

```
  private static final int DELAY_TIME = 6000;
  private static final int DEFAULT_MIN_TEMPERATURE = 0;
  private static final int DEFAULT_MAX_TEMPERATURE = 212;
  private static final String SETUP_FILE = "setup.bin";
  private int[] setupParameters;
  private volatile Thread timer;
```

The `PeriodicTask` class implements the `Runnable` interface so that the code that performs the periodic task can execute in its own thread. The class's `run()` method executes when `FormResponse`'s `init()` method calls the `start()` method of the `timer` thread.

In this example, the `run()` method contains an endless loop that waits for the number of milliseconds in `DELAY_TIME` to elapse, then writes the minimum and maximum settings to the console. In a real-world application, the `run()` method might perform tasks such as communicating with a temperature controller or monitor that uses the minimum and maximum values.

```
  private class PeriodicTask implements Runnable {

    public void run() {
      while (timer != null) {
        try {
          Thread.sleep(DELAY_TIME);
          System.out.print ("Minimum temperature = ");
          System.out.println (setupParameters[0]);

          System.out.print("Maximum temperature = ";
          System.out.println(setupParameters[1]);

        } catch (InterruptedException ex) {
        }
      }// end while (timer != null)
    } // end run()
  } // end PeriodicTask
```

The server calls `destroy()` after it takes a servlet out of service and all pending requests have either completed or timed out. A servlet should provide a `destroy()` method if it has acquired resources that won't otherwise be destroyed. In this example, the `destroy()` method stops the timer thread started by `PeriodicTask`'s `run()` method. Another reason to use a `destroy()` method is to save any data that the `init()` method might need next time and will otherwise be destroyed.

A call to `super.destroy()` calls the `destroy()` method of `Generic-Servlet` and writes a message to the log. The `destroy()` method then sets the `timer` thread to `null` and calls the thread's `interrupt` method. This generates an `InterruptedException` in `PeriodicTask`'s run method and terminates the thread.

```
public void destroy() {
  super.destroy();
  timer = null;
  timer.interrupt();
} // end destroy()
```

Servicing GET and POST Requests

The `doGet()` method calls the `sendSetupPage()` method, which returns a Web page with a form that enables users to view the current minimum and maximum temperature values and submit new ones. The parameters required for `sendSetupPage` are an `HttpServletResponse` object and either an error message to display on the page or `null` if there is no error message.

```
public void doGet( HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
  sendSetupPage (response, null);
} // end doGet()
```

The `doPost()` method receives and responds to data submitted on the form. On receiving values, the method checks to see if the values are within the specified ranges. If they are, the servlet returns a page that acknowledges receiving the data and stores the values in a file. If the values aren't acceptable, the servlet returns the form with an error message.

String variables hold the temperature values submitted on the form. Accepted values are stored as integers in the `setupParameters` array.

```
public void doPost(HttpServletRequest request,
  HttpServletResponse response)
    throws ServletException, IOException
{
  String minimumTemperature = null;
  String maximumTemperature= null;
  String errorMessage = null;
  int intMinimumTemperature = setupParameters[0];
  int intMaximumTemperature = setupParameters[1];
```

Calls to the `getParameter()` method of the `HttpServletRequest` object return the temperature values the client submitted on the form. The `Integer.parseInt` method converts the strings to integers. The server uses the integer values in determining whether the values are in the allowed ranges. An application that uses the values is also likely to want them in numeric form, rather than as strings.

For each value, the code tests to find out if the value is within the specified range. If not, an `errorMessage` string describes the problem.

```
  minimumTemperature =
    request.getParameter("minimum_temperature");
  if (minimumTemperature != null) {
    try {
      intMinimumTemperature =
        Integer.parseInt(minimumTemperature);
      if (intMinimumTemperature > 212 ||
        intMinimumTemperature < 0) {
        errorMessage = "Please try again: minimum
          temperature must be between 0 and 212.";
      }
    } catch (NumberFormatException e) {
      log("Invalid minimum temperature: ");
      log("minimumTemperature);

    }
  } // end if (minimumTemperature != null)
```

```
      maximumTemperature=
          request.getParameter("maximum_temperature");
      if (maximumTemperature!= null) {
        try {
          intMaximumTemperature =
              Integer.parseInt(maximumTemperature);
          if (intMaximumTemperature > 212 ||
              intMaximumTemperature < 0) {
            errorMessage = "Please try again: maximum
                temperature must be between 0 and 212.";
          }
        } catch (NumberFormatException e) {
          log("Invalid max. temperature: ";
          log(maximumTemperature);

        }
      } // end if (maximumTemperature!= null)
```

The code also checks to be sure that the minimum value submitted is less than the maximum. If not, an `errorMessage` string describes the problem.

```
      if (intMinimumTemperature >= intMaximumTemperature) {
        errorMessage = "Please try again: the minimum
          temperature must be less than the maximum
          temperature.";
      } // end if
```

The method then writes a Web page to the client. If the submitted values are acceptable, they're stored in the `setupParameters` array and a call to the `sendAcknowledgementPage()` method returns a Web page that acknowledges receiving the values. If the submitted values aren't acceptable, they aren't saved and a call to `sendSetupPage()` returns the form with the error message to advise the client to retry.

```
      if (errorMessage == null) {
        setupParameters[0] = intMinimumTemperature;
        setupParameters[1] = intMaximumTemperature;
        log("New minimum temperature: " +
          minimumTemperature);
        log("New maximum temperature: " +
          maximumTemperature);
        sendAcknowledgementPage(response);
      } else {
        sendSetupPage(response, errorMessage);
      }
    } // end  if (errorMessage == null)
```

Performing Tasks on Startup

The GenericServlet class includes an init() method that enables a servlet to perform tasks on startup. The init() method is called once, when the servlet starts, and is optional. In this example, init() calls the getSetupParameters() method to initialize the setup parameters and creates a thread that performs a periodic task.

```
public void init() throws ServletException {
  setupParameters = new int[2];
  getSetupParameters();
  timer = new Thread(new PeriodicTask());
  timer.start();
  System.out.println("The timer has started.");
  log("Timer started");
} // end init()
```

Saving and Retrieving Data in a File

The getSetupParameters() method retrieves the setup parameters from a file, if the file is available. Otherwise, the method uses the default values. A FileInputStream object attempts to read the parameters from the file whose name is stored in SETUP_FILE. The parameters are the first two values in the file.

```
private void getSetupParameters() {
  try {
     DataInputStream in = new DataInputStream
       (new FileInputStream(SETUP_FILE));
    int intMinimumTemperature = in.readInt();
    int intMaximumTemperature = in.readInt();
    setupParameters[0] = intMinimumTemperature;
    setupParameters[1] = intMaximumTemperature;
    try {
      in.close();
    } catch (IOException ex) {
    }
  } catch (FileNotFoundException ex) {
    log("Setup file not found");
    setupParameters[0] = DEFAULT_MIN_TEMPERATURE;
    setupParameters[1] = DEFAULT_MAX_TEMPERATURE;
  } catch (IOException ex) {
    log("Error reading from setup file", ex);
    setupParameters[0] = DEFAULT_MIN_TEMPERATURE;
    setupParameters[1] = DEFAULT_MAX_TEMPERATURE;
    }
  } // end getSetupParameters
```

The `saveSetupParameters()` method saves new setup parameters in the file whose name is stored in `SETUP_FILE`. A `FileOutputStream` object writes the values to the file. The parameters are the first two bytes in the file.

```
  private void saveSetupParameters() {
    try {
      DataOutputStream out = new DataOutputStream
        (new FileOutputStream(SETUP_FILE));
      out.writeInt(setupParameters[0]);
      out.writeInt(setupParameters[1]);
      out.flush();
      out.close();
    } catch (IOException ex) {
      log("Error writing to setup file", ex);
    }
  } // end saveSetupParameters
```

Acknowledging Received Form Data

The `sendAcknowledgementPage()` method sends a Web page to the client to acknowledge receiving submitted form data. The `doPost` method calls `sendAcknowledgementPage()` if the submitted data was accepted. A call to `saveSetupParameters()` stores the new data in a file. The `setContentType()` method of the `HttpServletResponse` object sets the `Content-Type` field of the HTML header in the returned. page. A `ServletOutputStream` object writes the Web page to the client.

The Web page displays the received values and also includes a hyperlink that enables the user to return to the setup page.

As in the previous TINI example, all quotation marks (") in the HTML code of the page being sent must be preceded by a back slash (\).

```
  private void sendAcknowledgementPage
    (HttpServletResponse response)
      throws IOException
  {
    saveSetupParameters();
    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();
    out.print ("<html>"
        + "<head>"
        + "<title> Form Data Received </title>"
        + "</head>"
        + "<body>"
        + "<h1> Form Data Received </h1>"
        + "<p>"
        + "The server has received the following
            settings:"
        + "</p>"
        + "<p>"
        + "Minimum temperature: ");
    out.print (setupParameters[0]);
    out.print ("</p>"
        + "<p> Maximum temperature: ");
    out.print (setupParameters[1]);
    out.print ("</p>"
        + "<p>"
        + "<a href=\"/servlet/FormResponse\">"
        + "Return to the temperature alarm setup page</a>"
        + "</p>"
        + "</body>"
        + "</html>");
  } // end sendAcknowledgementPage
```

Sending the Form

The `sendSetupPage()` method sends a Web page containing a form where the client can enter minimum and maximum temperature settings. The method uses the `HttpServeletResponse` object and the error message, if any, generated on examining previously submitted values.

The method calls `getSetupParameters()` to retrieve the values to display on the form. The `setContentType()` method of the `HttpServletResponse` object sets the `Content-Type` field of the HTML header in the returned page. A `ServletOutputStream` object writes the Web page to the client.

```
  private void sendSetupPage(HttpServletResponse
    response, String errorMessage)
      throws IOException
  {
    getSetupParameters();
    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();
    out.print ("<html>"
        + "<head>"
```

```
            + "<title>Temperature Alarm Setup</title>"
            + "</head>"
            + "<body>"
            + "<h1>Temperature Alarm Setup</h1>"
            + "<form method=POST action=
                \"/servlet/FormResponse\">"
            + "<table border>"
            + "<tr>"
            + "<td>Name</td>"
            + "<td>Value</td>"
            + "<td>Description</td>"
            + "</tr>"
            + "<tr>"
            + "<td> Minimum Temperature </td>"
            + "<td><input type=\"text\" name=
                \"minimum_temperature\" maxlength=3 value=");
        out.print (setupParameters[0]);
        out.print ("></td>"
            + "<td>Range 0 - 212 &deg;F</td>"
            + "</tr>"
            + "<tr>"
            + "<td> Maximum Temperature </td>"
            + "<td><input type=\"text\" name=
                \"maximum_temperature\" maxlength=3 value=");
        out.print (setupParameters[1]);
        out.print ("></td>"
            + "<td>Range 0 - 212 &deg;F</td>"
            + "</tr>"
            + "</table>");
        if (errorMessage != null)
          {
          out.print ("<p>" + errorMessage + "</p><p>");
          }
        out.print ("<input type=\"submit\"
            value=\"Submit\">"
            + "<input type=\"reset\" Value=\"Reset\">"
            + "</form></body>"
            + "</html>");
      } // end sendSetupPage()

    } // end FormResponse
```

8

E-mail for Embedded Systems

E-mail's primary use, of course, is to enable humans to send and receive messages over a network. But many embedded systems can make good use of e-mail as well. E-mail can be a convenient way for an embedded system to exchange information with humans or even communicate with other embedded systems with no human intervention at all.

For example, a security system can be programmed to send a message when an alarm condition occurs. Or a data logger might send a message once a day with the logger's readings for the previous 24 hours. In the other direction, an embedded system might receive e-mail containing new configuration settings or other commands, requests, or data.

E-mail has a couple of advantages over other methods of communication. Recipients can retrieve and read their messages whenever they want. And if the information isn't time-critical, the sender might find it easier or more efficient to place the information in an e-mail and send it off when conve-

6

Serving Web Pages with Dynamic Data

Chapter 5 showed how to use TCP and UDP to exchange messages containing application-specific data. Many standard application-level protocols also use TCP or UDP when exchanging information. One of the most popular of these is the hypertext transfer protocol (HTTP), which enables a computer to serve Web pages on request.

Because embedded systems almost always serve Web pages that contain dynamic, or real-time, information, this chapter begins with Rabbit and TINI examples that serve Web pages with dynamic content. Following the examples is an introduction to using HTTP and other protocols in serving Web pages.

Quick Start: Two Approaches

A Web browser such as Microsoft's Internet Explorer is a client application that uses HTTP to request Web pages from servers on the Internet or in a local network. The servers don't have to be PCs or other large computers. Even a small embedded system with limited memory can serve a page containing text and simple images, including pages that display real-time data and accept and act on user input.

A browser provides a user interface for requesting and displaying pages. The computers that request Web pages typically have full-screen displays, but for some applications, an embedded system with limited display capabilities can function as an HTTP client. If the requested pages are very simple, even a text-only display of a few lines might suffice. Or an embedded system might receive and process the contents of a Web page without displaying the page in a browser at all.

This chapter focuses on Web servers. With an Internet connection, a Web server can serve pages to any browser on the Internet. Or a server may be programmed to respond to requests only from specific IP addresses. A Web server in a local network may serve pages to selected computers or to any computer in the local network.

An embedded system that functions as a Web server generally has all of the following:

- Non-volatile memory to hold pages to be served.
- Support for TCP and IP. Requests for Web pages and the pages sent in response travel in the data portion of TCP segments.
- Support for HTTP. The server must be able to understand and respond to received requests for Web pages. The HTTP standard specifies the format for the requests and replies.
- A local-network or Internet connection. To serve pages on the Internet, the Web server must have an Internet connection. Any firewalls must be

configured so the system can receive HTTP requests, as described in Chapter 10.

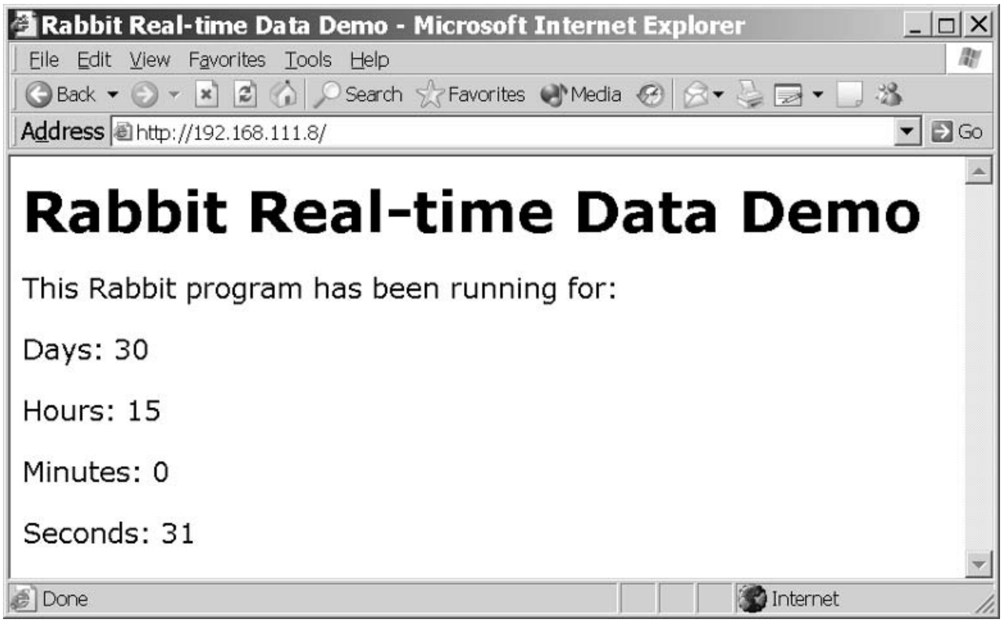
- One or more pages to serve. The Web pages are files or blocks of text that use a form of encoding called hypertext markup language (HTML). The HTML encoding specifies the formatting of text and images on the page, including text size and fonts and the positioning of text and other elements on the page. The HTML code may include links to images that appear on the page, as well as links to other pages or resources. In serving a Web page with dynamic content, the software must have a way of inserting the dynamic content as the page is being served.

A variety of protocols and technologies can work along with HTTP and HTML to enable a server to provide Web pages that contain real-time data and respond to user input. This chapter includes two approaches to serving real-time data, and Chapter 7 covers ways that Web servers can respond to user input.

Serving a Page with Dynamic Data

Many Web pages are static, where the information on the page doesn't change unless someone edits the page's HTML file and uploads the new file to the server. Static Web pages are useful for presenting product information, articles, or other information that remains constant. But most embedded systems have little use for static pages, other than possibly presenting a home page with links to other pages. An embedded system that functions as a Web server will almost certainly want to display real-time information such as sensor readings or other up-to-the-minute information about the processes or environments the system is controlling or monitoring.

This section shows how the Rabbit and TINI modules introduced in Chapter 3 can serve Web pages that display dynamic data. Dynamic, or real-time, data includes any data that can change over time and can be different each time the page is served. An obvious example is a counter that displays the number of times the page has been accessed. Dynamic data may also include sensor or switch readings and time and date information. The supporting code included with the Rabbit and TINI (and additional sources in the case


Rabbit Real-time Data Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.8/ Go
Done
Internet

Rabbit Real-time Data Demo

This Rabbit program has been running for:

Days: 30

Hours: 15

Minutes: 0

Seconds: 31

Figure 6-1: Both the Rabbit and TINI can serve pages that include dynamic, or real-time, data, such as the days, hours, minutes, and seconds displayed in these pages.

of Java servlets on the TINI) greatly reduces the amount of the programming required to serve Web pages with dynamic content.

The dynamic data served by the example applications in this chapter consists of a message that displays the amount of time the system or application has been up and running. Figure 6-1 shows an example page. The embedded system stores the number of days, hours, minutes, and seconds in variables. When serving the page, the server application inserts the current values of the variables in the appropriate places in the page. You can use the same techniques to create Web pages that display the current values of any variables in a system.

Although the result is the same, the Rabbit and TINI examples use different approaches to achieve the result. The Rabbit uses Server Side Include directives that instruct the server to insert the values of variables in the appropriate locations in the file being served. For the TINI, instead of storing the Web page in a separate file, the application creates the Web page as it's being sent, using a series of writes to send the page's contents to a TCP socket and inserting the values of variables in the designated locations in the page.

Rabbit Real-time Web Page

To serve its Web page, the Rabbit module uses HTTP functions and structures provided in Dynamic C to serve the Web page's file on request. The main program loop updates the time variables once per second.

Page Design

Listing 6-2 is the HTML code for Figure 6-1's Web page. The page uses HTML tags to advise the browser how to display the page's contents. Each tag consists of text enclosed by angle brackets (<>). The In Depth section of this chapter has more details about HTML tags and how to use them. For now, the relevant section of the code is the five lines that each begin with a paragraph tag (`<p>`).

A paragraph tag tells the browser to display the information that follows in a new paragraph. The first paragraph tag causes the browser to display the text, "This Rabbit program has been running for:".

Each of the four lines that follow contains a Server Side Include `#echo` directive that inserts the value of a variable on the page. A Server Side Include directive uses the same delimiters as an HTML comment. A comment, which is text that the browser ignores and doesn't display, is enclosed by `<!--` and `-->`. On receiving a page that contains an HTML comment, the browser displays the page the same as if the comment and its delimiters weren't present.

Another use for comment delimiters is to enable a page to specify Server Side Include (SSI) directives that the server executes before serving the page to the browser. Before serving a page containing an SSI directive, the server executes the directive and replaces the delimiters and the text between them with the result of executing the directive. If for some reason the server doesn't support the directive, the server ignores the directive and the browser treats the directive as a comment, which isn't displayed.

```
<html>

<head>
<title>Rabbit Real-time Data Demo</title>
</head>

<body>

<h1>Rabbit Real-time Data Demo</h1>

<p>This Rabbit program has been running for:</p>
<p>Days:    <!--#echo var="days"--></p>
<p>Hours:   <!--#echo var="hours"--></p>
<p>Minutes: <!--#echo var="minutes"--></p>
<p>Seconds: <!--#echo var="seconds"--></p>

</body>

</html>
```

Listing 6-2: On serving this Web page, the server retrieves the current values of “days”, “hours”, “minutes”, and “seconds” and inserts them in the page.

The `#echo` directive tells the server to replace the comment tag and its contents with the value of the named variable. For example, in the first directive, the server replaces `<!--#echo var="days"-->` with the value of the variable `days` on the server. If `days` equals 5, the browser receives and displays `Days: 5`.

The In Depth section of this chapter has more details about `#echo` and other Server Side Includes.

Serving the Page

The following is the complete application code the Rabbit requires to serve Figure 6-1’s Web page.

Initial Defines and Declarations

As in Chapter 5's examples, `TCPCONFIG` specifies a macro that sets a network configuration stored in the file *tcp_config.lib*. Your program must specify an appropriate macro for your system and network configuration.

The `#memmap` directive stores all C functions not declared as root in the extended memory area.

```
#define TCPCONFIG 1
#memmap xmem
```

The application requires the *dcrtcp.lib* library, which supports TCP/IP and related protocols, and the *http.lib* library, which supports HTTP.

The `#ximport` directive retrieves a file from the PC being used for project development, stores the file's length and contents in the Rabbit's extended memory, and associates a symbol (`index_html` in the example below) with the file's address in memory. Application code uses the symbol to locate the file and determine its length. The path in the `#ximport` statement must match the location of the file in your development PC. The file *index.shtml* contains the text in Listing 6-2.

```
#use "dcrtcp.lib"
#use "http.lib"
#ximport "c:/rabbitserver/index.shtml" index_html
```

Four variables store the values for the units of time the Web page will display.

```
unsigned long days;
unsigned long hours;
unsigned long minutes;
unsigned long seconds;
```

Dynamic C's HTTP server uses two structures, `HttpType` and `HttpSpec`, which contain information relating to the files the Web server serves.

The `HttpType` structure matches file extensions with file types and specifies a handler to use with files with the named extensions. When sending a file in response to an HTTP request, the server must identify the file type in the Content-Type field of the HTTP header in the response. The file types are Multipurpose Internet Mail Extension (MIME) types defined in RFC 2045 through RFC 2049.

The server in this application supports a single Web page with the extension *.shtml*, which is the conventional extension for files that use SSI directives. Dynamic C's handler for pages with SSI directives is `shtml_handler`.

The `HttpType` structure below associates *.shtml* with the MIME type *text/html*, which is a text file that uses HTML encoding. Other MIME types include *text/plain, image/jpeg*, and *audio/mpeg*. The server's default file (`"/"`) is associated with the first entry in the `Http_types` structure.

```
const HttpType http_types[] =
{
   { ".shtml", "text/html", shtml_handler}
};
```

The `HttpSpec` structure contains information about the files, variables, and structures that the Web server can access. Each entry in the structure has seven parameters, though not all entry types use all of the parameters. The structure in this example has entries for two files and four variables:

```
const HttpSpec http_flashspec[] =
{
 { HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL, NULL},
 { HTTPSPEC_FILE, "/index.shtml", index_html, NULL, 0,
   NULL, NULL},

 { HTTPSPEC_VARIABLE, "days", 0, &days, INT32, "%d",
   NULL},
 { HTTPSPEC_VARIABLE, "hours", 0, &hours, INT32, "%d",
   NULL},
 { HTTPSPEC_VARIABLE, "minutes", 0, &minutes, INT32, "%d",
   NULL},
 { HTTPSPEC_VARIABLE, "seconds", 0, &seconds, INT32, "%d",
   NULL},
};
```

The `HTTPSPEC_FILE` entries associate the symbols defined in `#ximport` statements with the names of files that browsers may request from the server. These are the parameters for an `HTTPSPEC_FILE` entry:

Type. Indicates whether the entry is for a file, variable, or function. `HTTPSPEC_FILE` specifies that the entry is for a file.

Name. Names a file the Web server can access. This example has one file, *index.shtml*, with two entries to enable browsers to request the file by name (`"index.shtml"`) or as the default file to serve when no name is specified (`"/"`).

Data. Specifies the file's physical address. Both `HTTPSPEC_FILE` entries point to `index_html`, where the file *index.shtml* is stored.

Addr. Unused (`NULL`) for files.

Vartype. Unused (zero) for files.

Format. Unused (`NULL`) for files.

Realm. Names an `HttpRealm` structure that identifies a name and password required to access the file. `NULL` if unused.

The four `HTTP_VARIABLE` entries specify variables for the different units of time. These are the parameters for an `HTTP_VARIABLE` entry:

Type. Indicates whether the entry is for a file, variable, or function. `HTTPSPEC_VARIABLE` specifies that the entry is for a variable.

Name. Provides the name of a variable the Web server can access. The server's Web page displays the values of four variables: `"days"`, `"hours"`, `"minutes"`, and `"seconds"`.

Data. Unused (zero) for variables.

Addr. A short pointer to the variable.

Vartype. The type of variable. The options are 8-bit integer (`INT8`), 16-bit integer (`INT16`), 32-bit integer (`INT32`), 16-bit pointer (`PTR16`), and 32-bit floating-point value (`FLOAT32`). The `PTR16` type is useful for displaying strings.

Format. The `printf` specifier to use when displaying the variable. The specifier `%d` causes the variable to display as a decimal value.

Realm. Identifies a name and password to access the variable. `NULL` if unused.

The main() Function

The application's `main()` function begins by declaring variables related to time, calling `sock_init()` to initialize the TCP/IP stack, and calling `http_init()` to initialize the Web server. Calling `tcp_reserveport()` to reserve port 80 for the Web server is optional but can improve performance in two ways: by allowing a socket to be established even if the server can't exchange data yet and by shortening the waiting period for closing a socket.

```
main()
{
   unsigned long start_time;
   unsigned long total_seconds;

   sock_init();
   http_init();
   tcp_reserveport(80);
```

Dynamic C's `SEC_TIMER` variable contains the number of seconds since midnight on the morning of January 1, 1980. The program uses this value to measure how long the program has been running, beginning with an initial count stored in `start_time` when the program begins running:

```
   start_time = SEC_TIMER;
```

The program's endless `while` loop has two responsibilities. It calls `http_handler()`, which is required periodically to parse received requests and pass control to `shtml_handler()` or another handler specified in the `HttpType` structure. And a costatement updates the time variables once per second. (Chapter 3 introduced Dynamic C's costatements.)

When one second has elapsed, as specified in the `waitfor(DelaySec(1))` statement, the program calculates the number of seconds it has been running by subtracting the `start_time` value from the current value of `SEC_TIMER`.

The program then divides the number of seconds into days, hours, minutes, and seconds:

To find the number of days, take the integer result of `total_seconds` divided by the number of seconds per day (86,400).

To find the number of hours, divide `total_seconds` by the number of seconds per hour (3600) to get the total number of elapsed hours. Eliminate any full days with modulus division by the number of hours per day (24).

To find the number of minutes, divide `total_seconds` by the number of seconds per minute (60) to get the total number of elapsed minutes. Eliminate any full hours with modulus division by the number of minutes per hour (60).

To find the number of seconds excluding full minutes, use the result of modulus division of `total_seconds` by the number of seconds per minute (60).

```
    while (1) {
       http_handler();
       costate {
         waitfor(DelaySec(1));
         total_seconds = SEC_TIMER - start_time;
         days = total_seconds /86400;
         hours = (total_seconds /3600) % 24;
         minutes = (total_seconds /60) % 60;
         seconds = total_seconds % 60;
       }

       //Code to perform other tasks can be placed here.

    } // end while(1)

 } // end main()
```

Dynamic C's HTTP server and SHTML handler code manage the serving of the Web page, including accepting requests to connect, returning the requested pages or other HTTP responses as appropriate, and closing connections.

In a real-world application, the main loop would probably perform other tasks as well. The costatement ensures that other tasks will get their turn even while waiting for the costatement's delay timer to time out.

Accessing the Web Server

When the Rabbit is running this code, you can request its Web page by entering the module's IP address in a browser's Address text box:

```
http://192.168.111.7
```

or by specifying the IP address and Web page:

```
http://192.168.111.7/index.shtml
```

If a domain name is assigned to the IP address, you can use that as well to request the page. On receiving a request for a page, the Rabbit's HTTP server appends an appropriate HTTP header to the top of the requested file and writes the header and file to the socket that requested it. The SHTML handler replaces the `#echo` directives on the page with the current values of `days, hours, minutes` and `seconds`. And the browser that requested the file displays Figure 6-1's Web page, which contains the time values.

Refreshing the page in the browser updates the displayed time. To update the display automatically at intervals, see Refreshing Pages Automatically later in this chapter.

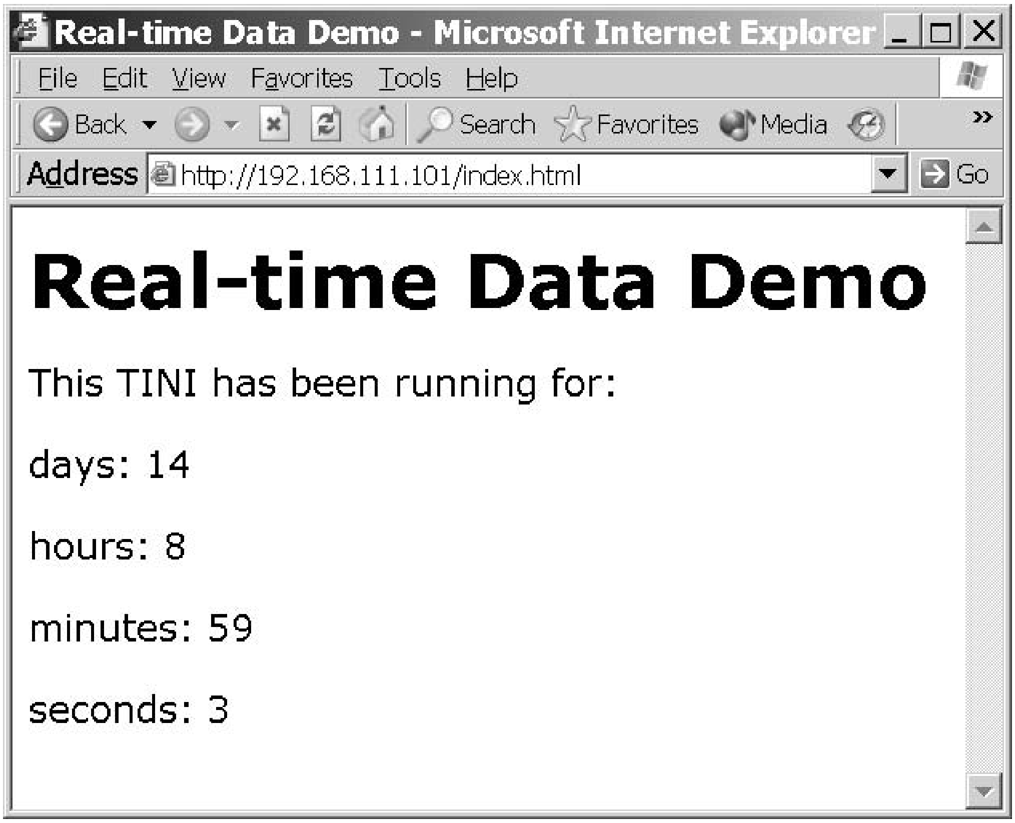
TINI Real-time Web Page

To use a TINI to serve Web pages with dynamic content, you have a few choices. Your first thought might be to use the `HttpServer` class provided with the TINI's operating system. However, this built-in Web server can only serve static pages. Serving dynamic data would require changing the data in the stored pages whenever the content changes. It's more efficient to retrieve the dynamic data on request and insert it in the page as it's being served.

Another option is to install and run a server program that supports Java servlets. A servlet is a software component that can respond to user input and generate dynamic content for Web pages. In most cases, servlets are the most effective and time-saving way to enable a Web server to serve dynamic content. Chapter 7 has more about servlets and how to use them.

A third option is to write a basic Web server that uses the `ServerSocket` class and adds dynamic content as it serves its pages. For some low-volume applications that serve one or a few pages, this kind of home-brewed server can do the job without adding too much complexity. The example in this chapter uses the `ServerSocket` class to create a basic server that serves a page that displays the amount of time the TINI has been up and running. Whether or not you decide to use this approach, the code in this application is interesting as a demonstration of the responsibilities of a Web server.

The Web server responds to requests to connect to a specific port. When a connected host sends an HTTP request for a supported page, the server cal-


Real-time Data Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.101/index.html
Go

Real-time Data Demo

This TINI has been running for:

days: 14

hours: 8

minutes: 59

seconds: 3

Figure 6-3: This Web page served by a TINI shows how long the TINI has been running since it booted up.

culates the values of variables the page contains, writes the contents of the page to the socket, and closes the socket.

Serving the Page

Figure 6-3 shows the Web page, and Listing 6-4 is the source code for the page as received by a browser. The program code below is an application that serves Figure 6-1's Web page. The code is very similar to the TcpServer example in Chapter 5, with the addition of code that parses requests and returns the Web page or error code.

```
<html>

<head>
  <title>Real-time Data Demo </title>
</head>

<body>
  <h1>Real-time Data Demo</h1>

  <p> This TINI has been running for:</p>
  <p>days: 14 </p>
  <p>hours: 8 </p>
  <p>minutes: 59 </p>
  <p>seconds: 3 </p>

</body>

</html>
```

Listing 6-4: The HTML code for Figure 6-3's Web page.

Imports and Initial Declarations

The code imports `java.net` classes for networking functions and `java.io` classes to support input and output functions. The TINI-specific `TINIOS` class includes an `uptimeMillis()` method the application uses to retrieve the number of milliseconds the TINI has been up and running.

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

The `RealTimeWebPage` class implements the `Runnable` interface so that the code that does the network communications can execute in its own thread. This leaves the main thread free to do other things.

```
public class RealTimeWebPage implements Runnable {

  private ServerSocket server;
  private int readTimeout;
  private Thread serverThread;
  private volatile boolean runServer;
```

The main() Method

The class's `main()` method sets `localPort` to the port number clients will connect on and sets `readTimeout` to the number of milliseconds the server will wait to receive data after a remote host connects. Port 80 is the default port for HTTP requests. The timeout is expressed in milliseconds. The `RealTimeWebPage` object `server` uses the `localPort` and `readTimeout` values.

```
public static void main(String[] args) throws
    IOException {
  int localPort = 80;
  int readTimeout = 5000;

  RealTimeWebPage server =
      new RealTimeWebPage(localPort, readTimeout);
```

An endless loop executes while waiting for connections. The thread spends its time sleeping, but could perform other tasks.

```
  while (true){
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.print("InterruptedException: ");
      System.out.println(e.getMessage() );
    }
  } // end while(true)
} // end main
```

Initializing the Server

The constructor for the `RealTimeWebPage` class creates a thread to handle connection requests. The constructor's two parameters are the `localPort` and `readTimeout` values set in `main()`.

```
public RealTimeWebPage(int localPort, int readTimeout)
    throws IOException {
```

A `ServerSocket` object (`server`) listens for connection requests at `localPort`, and on receiving a request, creates a socket object.

```
  server = new ServerSocket(localPort);
  System.out.println("The server is listening on port "
      + localPort + ".");
```

The readTimeout variable used by the run() method below is assigned the value of the readTimeout parameter.

```
      this.readTimeout = readTimeout;
```

A separate thread (serverThread) handles connection requests. Setting the thread's setDaemon method true creates the thread as a Daemon thread. The JVM exits when there are no user (non-Daemon) threads running. Calling the start() method calls the thread's run() routine.

```
      serverThread = new Thread(this);
      serverThread.setDaemon(true);
      serverThread.start();
    } // end RealTimeWebPage constructor
```

Waiting for Connection Requests

Calling serverThread's start() method causes the thread's run() method to execute. The run() method accepts connections and calls a method to handle each connection.

```
    public void run() {
```

An endless loop runs until the runServer variable is false, which occurs on an exception or if the class's stop() method sets runServer false.

```
      runServer = true;
      while (runServer) {
        try {
```

On accepting a connection request, the server's accept() method creates a socket for exchanging data with the connected host. The class's handleConnection() method manages communications with the socket.

When handleConnection() returns or if there is an exception, the method closes the socket to release any resources used by it. If there is an exception when attempting to close the socket, no action needs to be taken.

If an exception occurs while attempting to accept a connection, runServer is set to false to stop the thread.

```
          Socket socket = server.accept();

          try {
            handleConnection(socket);
          } catch (IOException e) {
```

```
          System.out.print("IOException: ");
          System.out.println(e.getMessage());
        } finally {
          try {
            socket.close();
          } catch (IOException e) {
          }
        }
      } catch (IOException e) {
        runServer = false;
        System.out.print("IOException: ");
        System.out.println(e.getMessage());
      }
    } // end while(runServer);
  } // end run
```

Stopping the Server

The `stopServer()` method provides a way to stop the server under program control by setting `runServer` `false` and closing the socket.

```
  public void stopServer() {
    runServer = false;
    try {
      server.close();
    } catch (IOException e) {
    }
  } // end stopServer
```

Handling a Connection

The `handleConnection()` method handles a single connection with a remote host. The socket timeout is set to the `readTimeout` value set in the `main()` routine.

```
  private void handleConnection(Socket socket)
      throws IOException {

    System.out.println("Connected to " + socket);
    socket.setSoTimeout(readTimeout);
```

An `InputStream` object reads data from the remote host, and a `PrintStream` object writes to the remote host.

```
    InputStream in = socket.getInputStream();
      PrintStream out =
          new PrintStream(socket.getOutputStream());
```

The class's `processRequest()` method reads the received data and returns the requested web page or an error page.

When `processRequest` returns, the `Printstream` object's `checkError()` method flushes the output stream and returns true if an `IOException` other than `InterruptedIOException` has occurred or if the Printstream object's `setError()` method has been invoked.

```
      try {
        processRequest(in, out);
        if (out.checkError()) {
          System.out.println("An error occurred while
              sending a web page.");
        } else {
          System.out.println("A response was sent.");
        }
      } catch (InterruptedIOException e) {
        System.out.print("InterruptedIOException: ");
        System.out.println(e.getMessage());
        System.out.print("The connection timed out after
            receiving no data for : ");
        System.out.print(readTimeout / 1000);
        System.out.println(" seconds.");
      }
    } // end handleConnection
```

Processing a Request

The `processRequest()` method reads an incoming request and takes appropriate action.

```
    private void processRequest(InputStream in,
          PrintStream out) throws IOException
    {
```

The first step is to read the first four bytes from the `PrintStream` object.

```
      int b1 = in.read();
      int b2 = in.read();
      int b3 = in.read();
      int b4 = in.read();
```

This server supports GET requests only. The HTTP standard requires GET to be upper case, followed by a space. If the received bytes equal GET, followed by a space, the code reads any bytes that follow and stores them in the `StringBuffer` object `requestBuffer`. Reading the input stops on detecting a space character, a carriage return (`\r`), line feed (`\n`), or an end of file marker (-1).

```
    if (('G' == b1) && ('E' == b2) && ('T' == b3) &&
        (' ' == b4)) {
      StringBuffer requestBuffer = new StringBuffer();
      int b = in.read();
      while ((b != -1) && (b != ' ') && (b != '\r') &&
          (b != '\n')) {
        requestBuffer.append((char)b);
        b = in.read();
      }
```

The `StringBuffer` object is converted to a `String` to enable examining its contents. The server accepts requests for the default page (indicated by "`/`") or for the file */index.html*. If there is a match with either of these, the class's `sendWebPage` routine returns the real-time Web page to the requesting host. If there isn't a match, a call to the `sendErrorPage()` method returns error code 404 and an error message to the requesting host.

```
      String requestedPage = requestBuffer.toString();
      String defaultPage = "/";
      String indexPage = "/index.html";

      if ((requestedPage.equals(defaultPage)) ||
          (requestedPage.indexOf(indexPage) != -1) ) {

        sendWebPage(out);

      } else {
        sendErrorPage(out, "404 Not Found");
      }
```

If "GET " wasn't received, the program checks to see if -1 was returned. If so, the input stream is closed, so there is nothing to return to the remote host. For any received data besides "GET " or -1, a call to the `sendErrorPage()` method returns error code 501 and the error message "Not Implemented" to the requesting host.

```
    } else {
      if ((b1 | b2 | b3 | b4) != -1) {
        sendErrorPage(out, "501 Not Implemented");
      }
    } // end if ('G'==b1||'E'==b2||'T'==b3||' '==b4)
  } // end processRequest
```

Sending the Web Page

The `sendWebPage()` method uses the `PrintStream` object to send the page containing real-time data.

```
private void sendWebPage(PrintStream out)
    throws IOException {
```

The page begins with the response's start line and HTML headers. A blank line (`\r\n`) after the HTTP header indicates the end of the header. The In Depth section of this chapter has more about these elements of a response.

```
out.print("HTTP/1.0 200 OK\r\n"
          + "Content-Type: text/html\r\n"
          + "\r\n");
```

A call to the TINIOS class's `uptimeMillis()` method returns the number of milliseconds that have elapsed since the TINI booted up. The page displays the time in days, hours, minutes, and seconds. To obtain the total number of seconds, divide `uptimeMillis()` by 1000.

```
long totalSeconds = TINIOS.uptimeMillis()/1000;
```

For the number of days, divide by the number of seconds per day:

```
long days = totalSeconds / 86400;
```

For the number of hours, divide by the number of hours per day and use modulus division to subtract any full days:

```
long hours = (totalSeconds /3600) % 24;
```

For the number of minutes, divide by the number of minutes per day and use modulus division to subtract any full hours:

```
long minutes = (totalSeconds / 60) % 60;
```

For the number of seconds, use modulus division to subtract any full minutes:

```
long seconds = totalSeconds % 60;
```

A series of `out.print` statements sends the page's contents to the requesting host. The page consists of blocks of static text, plus the values of the four variables inserted at the appropriate locations in the page. The `out.print` statements use the + operator to concatenate multiple String constants. This method keeps the code readable while limiting the number of writes to the

`PrintStream` object. The four variables each have their own `out.print` statements, however. This is because concatenating String variables uses large amounts of memory and processing power in the TINI. String constants don't have this effect, so concatenating these has no ill effects.

```
      out.print("<html>"
               + "<head> <title> "
               + "Real-time Data Demo "
               + "</title> </head>"
               + "<body>"
               + "<h1> Real-time Data Demo</h1>"
               + "<p> This TINI has been running for:</p>"
               + "<p> days: ");
      out.print(days);
      out.print(" </p>"
               +"<p>"
               + "hours: ");
      out.print(hours);
      out.print(" </p>"
               + "<p>"
               + "minutes: ");
      out.print(minutes);
      out.print(" </p>"
               + "<p>"
               + "seconds: ");
      out.print(seconds);
      out.print (" </p>"
                + "</body>"
                + "</html>");
   } // end sendWebPage()
```

Sending an Error Page

If the connected host sends a request for a non-existent page or a request other than GET, the `sendErrorPage` method uses a series of `out.print` statements to return an error code and a page that displays an error message. The `errorMessage` parameter contains the message.

```
   private void sendErrorPage(PrintStream out,
       String errorMessage) throws IOException
   {
```

The first text sent is the response's start line containing the error message and Content-Type header, followed by the required blank line.

```
      out.print("HTTP/1.0 ");
      out.print(errorMessage);
      out.print("\r\n"
                + "Content-Type: text/html\r\n"
                + "\r\n");
```

Another series of `out.print` statements then sends a Web page that displays the error message.

```
      out.print("<html>"
                + "<head><title>");
      out.print(errorMessage);
      out.print("</title></head>"
                + "<body>"
                + "<h1>");
      out.print(errorMessage);
      out.print("</h1>"
                + "</body>"
                + "</html>");
    } // end sendErrorPage
  } // end RealTimeWebPage
```

Running the Server

As in the Rabbit example, you can request the TINI's Web page by entering its IP address, IP address and file name, or domain name (if available) in a browser's Address text box.

These examples show two different but equally useful ways to serve Web pages with dynamic data. Chapter 7 expands on this topic by showing two ways to create Web pages that can respond to user input in addition to displaying dynamic content.

In Depth: Protocols for Serving Web Pages

The examples in this chapter showed how Web browsers use the hypertext transfer protocol (HTTP) to request Web pages, and the Web pages themselves are encoded using the hypertext markup language (HTML). In addition, some pages use server-side include (SSI) directives to enable a Web page to display dynamic data or to add other capabilities not available with HTML alone.

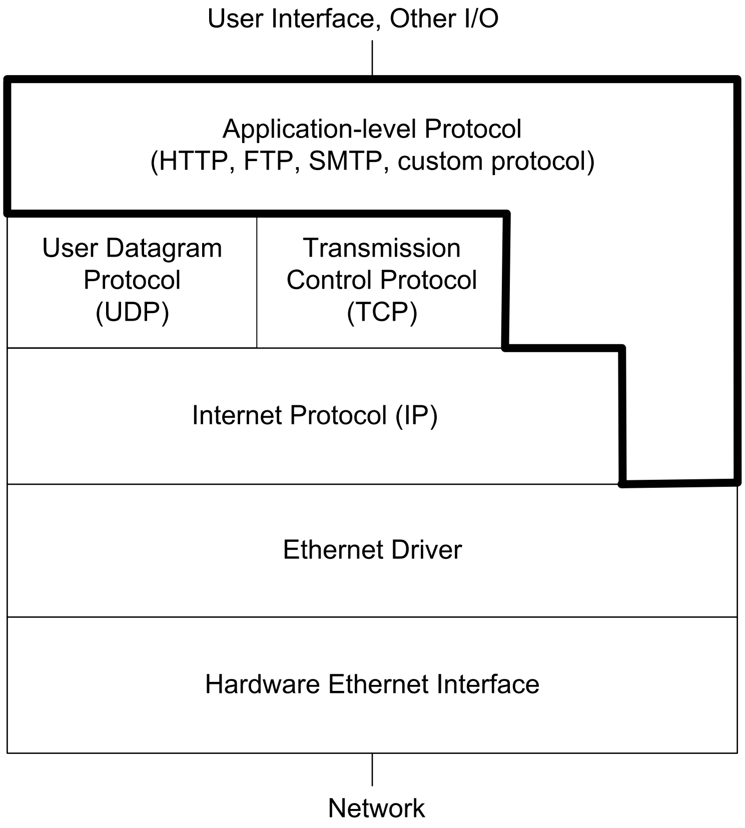

User Interface, Other I/O
Application-level Protocol (HTTP, FTP, SMTP, custom protocol)
User Datagram Protocol (UDP)
Transmission Control Protocol (TCP)
Internet Protocol (IP)
Ethernet Driver
Hardware Ethernet Interface
Network

Figure 6-5: HTTP is an application-level protocol layer communicates with the Ethernet driver and either a UDP or TCP layer or the application layer.

This section has more details about HTTP, HTML, and SSI, with the focus on how embedded systems can use each in serving pages with dynamic content.

Using the Hypertext Transfer Protocol

HTTP is one of many standard application-level protocols used in network communications. Figure 6-5 shows the location of HTTP in a network protocol stack. Although in theory an HTTP communication can use any reliable protocol to reach its destinations on a network, in practice just about all network stacks pass HTTP communications through TCP and IP layers. An application that uses HTTP may be a Web browser, which requests Web pages, or a Web server, which returns Web pages on request.

Anyone who has browsed the Internet has used HTTP. When a browser sends a request for a Web page onto the network, the request contains a URL that identifies the location and file name of the page. Chapter 4 described how a network uses the information in the URL to determine where to route a communication.

On learning the IP address that is hosting the desired Web page, the client requests to open a TCP connection with the computer at that address. By default, Web servers serve pages on port 80. If a server is using a different port number, the URL specifies the number, as explained in Chapter 4. When the connection has been established, the browser sends a message containing an HTTP request for a page, and the receiving computer responds by serving, or sending, the Web page to the requesting computer over the TCP connection.

A benefit of using Web pages to provide information is that the browser interface is universal. If you place a Web server on the Internet, anyone with a browser and an Internet connection can view the server's pages. Search engines make it possible for users to find your page even if they don't know the IP address or domain name. Web pages don't have to be on the Internet, however. You can make a page available only within a local network. If desired, you can also restrict access by specifying what IP addresses can access a page or by requiring a password to access the page. In any case, you don't have to limit communications to users who are using specific hardware or software.

As the examples in Chapter 7 show, a server can also receive information from a browser. A Web page can enable users to send information to the computer that is serving a page, and the computer can use this information for any purpose.

HTTP Versions

HTTP version 1.1 is specified in *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1.* RFC1945 contains the previous versions, HTTP 1.0 and 0.9.

Version 1.1 adds capabilities for conserving network bandwidth, improving security and error notification, enabling clients to specify preferred languages or character sets, and allowing more flexible buffering by dividing data into chunks.

Many embedded systems serve small and simple Web pages. These systems may gain little benefit in supporting HTTP 1.1 and thus may use 1.0 for simplicity. HTTP 1.0 servers must also respond appropriately to requests from 0.9 clients. A browser that supports HTTP 1.1 should have no trouble communicating with a 1.0 server. Dynamic C's HTTP server complies with HTTP 1.0. The Tynamo Web server used in Chapter 7's TINI examples implements the required elements in HTTP 1.1.

Probably the main reason an embedded system might use HTTP 1.1 is its support for persistent connections, which can reduce the number of connections the server must open and close. With HTTP 1.0, each request requires a new connection. If a client requests a Web page that contains several links to images, the request for the page as well as each request for an image requires its own connection, which in turn requires the server and client to do the handshaking to open and close each connection. Requesting multiple pages within a short time also requires a new connection for each page. In contrast, with HTTP 1.1, the default behavior is persistent connections, where a connection is left open until either the client or server determines that the communication is complete or the server closes the connection after a period of no activity.

The RFC documents spell out the minimum capabilities that an HTTP server must have. The requirements vary with the HTTP version.

Elements of an HTTP Message

An HTTP message consists of an initial request or status line, optional message headers, a blank line, and an optional entity body. (HTTP 0.9 doesn't support status lines or headers.)

HTTP supports two types of messages, requests and responses. A client sends a request to ask a server for a resource, and the server returns a response containing the resource or status information.

Requests

An HTTP 1.0 request must contain at least two lines: the request line and a blank line. Some requests also have one or more message headers between the request line and the blank line, and some requests have an entity body following the blank line. Here is an example request for the file */index.html* from the host at *www.example.com*:

```
GET /index.html HTTP/1.0\r\n
Host: www.example.com\r\n
Accept: */*\r\n
Connection: close\r\n
\r\n
```

Each line in the request terminates in `\r\n`, which is a pair of escape sequences equivalent to a carriage return, or return to the beginning of the line (`\r`), followed by a line feed, or drop to a new line immediately below the current line (`\n`). Escape sequences provide a way of expressing text formatting commands such as these using plain text.

The Request Line

In the following request line:

```
GET /index.html HTTP/1.0
```

`GET` is a method that tells the server that the client is requesting a resource from the server. The `HTTP/1.0` in the request line tells the server that the highest version of HTTP the client supports is 1.0.

`/index.html` is the name and path of the resource the client is requesting from the server. The "`/`" indicates that the file is in the server's root directory. The server's root directory may be, but doesn't have to be, the same as the root directory in the system's file system. For example, a server may define its root directory as */http-root*. Clients can then access files in */http-root* and its subdirectories (such as */http-root/images*), but not files in other directories under the system's root directory (such as */private*).

A GET request often contains only the file name and path, but an HTTP 1.1 server must also accept a request that contains a full URL such as this:

```
GET http://www.Lvr.com/index.html HTTP/1.1
```

On receiving a page that includes images, the client typically sends a GET request for each image.

In addition to the GET method, HTTP 1.0 and later define the HEAD and POST methods (Table 6-1). HEAD is similar to GET except that the server returns only the headers it would send in responding to a GET request for the resource, but not the resource itself. The POST method enables a client to send data to a resource on the server. The server passes the data received in the message body to the program, process, or other resource specified in the request line. The named resource uses the data. A common use for POST is to enable users to send data entered on a form to a CGI program, which processes the data and sends a response to the client. (Chapter 7 has more about CGI.) But a POST request can specify any resource, and the resource can use the data in any way.

The HTTP 1.1 standard says that all general-purpose servers must at minimum support the GET and HEAD methods.

HTTP 1.1 defines additional methods. One that embedded systems might use is PUT, which like POST, enables the client to send data to the server. But instead of naming a resource to receive the message body's data, a PUT request names a file or other entity where the server should store the message body's data. PUT can be useful for file transfers, where the request line names the file on the server where the server should store the received data.

HTTP 0.9 supports only the GET method, and the request line includes only the request and the URL, not the HTTP version. If no HTTP version is specified, the server should assume it's version 0.9.

Methods specified in requests must be upper case and followed by a space.

Headers

A message may contain headers between the request line and the blank line. A header can contain additional information about the request, such as the number of data bytes in the message body, or more general information, such as a date. Headers generally have the following format:

header_name: *data*

Table 6-1: Selected HTTP Methods Used in Requests

Method	HTTP Version Introduced In	Description
GET	0.9	Retrieve the specified Web page or other information
HEAD	1.0	Retrieve only the HTTP headers for the specified information (not the message body)
POST	1.0	Pass the information in the message body to the resource identified in the request line
PUT	1.1	Store the information in the message body in the file or other entity identified in the request line

The HTTP standard specifies valid header names and what data each header provides. For example, a client might include an Accept header in a request for a Web page to inform the server of what types of content the client can accept. In this example, the client accepts images in *.gif* and *.jpeg* formats:

```
 Accept: image/gif, image/jpeg
```

This means that the client accepts all media types:

```
Accept: */*
```

If an HTTP 1.0 request includes data in the message body, the standard requires a Content-Length header that specifies the number of bytes in the message body:

```
 Content-Length: 256
```

HTTP 1.1 also requires requests that include data in the message body to transmit the content length, but supports additional ways of doing so.

This HTTP 1.1 header:

```
  Connection: close
```

indicates that the current connection is not persistent and should be closed after the response is sent. An HTTP 1.1 host that doesn't support persistent connections must send this header with every connection. HTTP 1.0 hosts don't support persistent connections or this header, which they can ignore if received.

An Authorization header enables a client to send authentication information such as a user name and password, usually after receiving a response with a WWW-Authenticate header, as described in Chapter 10.

An HTTP 1.1 request must include a Host header in each request. The Host header specifies the Internet host name (such as *www.Lvr.com*) of the resource being requested. The requirement for a Host header was added in the hope of conserving IP addresses by making it easier for a single IP address to support multiple host names. For example, a server might host both *www.example.com* and *www.Lvr.com* at the same IP address. On receiving a GET request for a default page, the server's HTTP software can examine the Host header to find out which host's page the client is requesting. Without the Host header, each host name needs its own IP address.

When a request is directed to a port other than the protocol's default port, the Host header includes this information as well:

```
Host: www.Lvr.com:5501
```

If a server doesn't have an Internet host name, the request must include a Host header with an empty value. The HTTP 1.1 standard says that when an HTTP 1.1 server receives an HTTP 1.1 request that doesn't include a Host header, the server must return a status code of 400 (Bad Request). HTTP 1.0 doesn't support the Host header, so HTTP 1.0 requests don't include it and 1.0 servers can ignore it if received.

The Message Body

The message body contains data the client is providing to the server, such as the data in a POST or PUT request.

Responses

On receiving an HTTP request, the server returns a response. An HTTP 1.0 response must contain at least two lines: a status line and a blank line to indicate the end of the headers. Some responses also have one or more message headers between the status line and the blank line, and some responses have a message body following the blank line. HTTP 0.9 servers return the message body only. Here are the status line and headers sent in reply to a request for a Web page:

```
HTTP/1.0 200 OK\r\n
Date: Wed, 09 Jul 2003 12:02:51 GMT\r\n
Content-Type: text/html\r\n
Content-Length: 432\r\n
\r\n
```

As with requests, each line in the response terminates with a carriage return and line feed (`\r\n`).

The Status Line

The status line contains the version of HTTP supported by the server and a status code and text phrase that give the result of the request. On success, this is the status line from an HTTP 1.0 server:

```
HTTP/1.0 200 OK
```

If the client requests a non-existent file, the response is this:

```
HTTP/1.0 404 Not Found
```

The HTTP standard includes a series of status codes and suggested text phrases to use with them.

Response Headers

A response can use headers to send additional information about a response or to return general information about the message or connection.

The HTTP 1.0 standard says that the header for a response that contains a message body should include a Content-Length field that gives the length of the message body in bytes:

```
Content-Length: 14092
```

If the Content-Length field isn't included, the closing of the connection determines the length of the message body.

A Date field indicates when the response message was generated (not when the Web page or other resource was created):

```
Date: Thu, 08 May 2003 02:45:58 GMT
```

If possible, servers should include a Date field in responses. However, a server that doesn't have a reasonably accurate clock must not include a Date field. The preferred format for the contents of the Date field is the *rfc1123-date* format specified in the HTTP 1.1 standard and RFC 1123. This format uses a fixed-length field for each element in the date. The example above uses this format. A computer that receives a response without a date can add a date to a received response if needed.

When a client requests a password-protected resource, the server can return a WWW-Authenticate header to request the client to provide a user name, password, or other authentication information before gaining access to the resource. Chapter 10 has more about using this header.

The HTTP standards specify the headers supported by each HTTP version.

Message Body

The message body contains any data the response wants to return to the client. In a response to a GET request, the message body contains the requested Web page or other resource.

Using HTTP with Other Client Applications

The main use for HTTP is for communicating with Web browsers, but other applications can send HTTP requests as well. For example, to retrieve and store information from a Web page, you could write an application that requests a page and then searches the response for desired information. A timer routine can trigger a page retrieval periodically.

Inside the Hypertext Markup Language

Related to HTTP is the Hypertext Markup Language (HTML) used in Web pages. HTML defines codes that specify how text and images appear on a Web page.

The HTML specification is available from the World Wide Web Consortium (W3C), at *www.w3.org*. The members of W3C are organizations interested in developing common protocols for the World Wide Web. HTML version 4.01 was released in 1999. Rather than continuing to update the

HTML specification, W3C has switched development to Extensible HTML (XHTML), a more flexible and powerful language whose roots are in HTML. For basic Web pages, you don't need anything beyond what's available in HTML. It's possible to create HTML Web pages that also comply with the XHTML specification.

Creating HTML Pages

You can create HTML pages using any text editor, including Windows Notepad. Or you can use a Web-design application such as Macromedia Inc.'s Dreamweaver, which enables you to create pages visually using toolbars and menus to add page elements and formatting. The application inserts the appropriate HTML code as needed. Some embedded systems serve very basic pages that require little in the way of fancy formatting or other features. When this is the case, using a text editor to create the pages is a reasonable choice. But even if you use a specialized application to create your pages, a little knowledge of HTML can be useful in ferreting out the inevitable problems that crop up.

The conventional extension for HTML-encoded files is *.html* or *.htm*.

This book provides only the most basic introduction to HTML. For more detail, refer to the specification or a book such as *HTML 4 for the World Wide Web* by Elizabeth Castro (Peachpit Press).

Using Tags

Figure 6-6 shows a basic Web page that displays text and an image. Listing 6-7 is the file that contains the HTML code for the page. HTML tags specify the text formatting and placement of the image. Each tag contains an HTML element enclosed in angle brackets (<>). Some elements have one or more required or optional attributes, which provide additional information about the element. For example, in this tag:

```
<input type="submit" value="Submit">
```

the element is `input`, and `type` and `value` are attributes that name the input type and the text the input button displays.

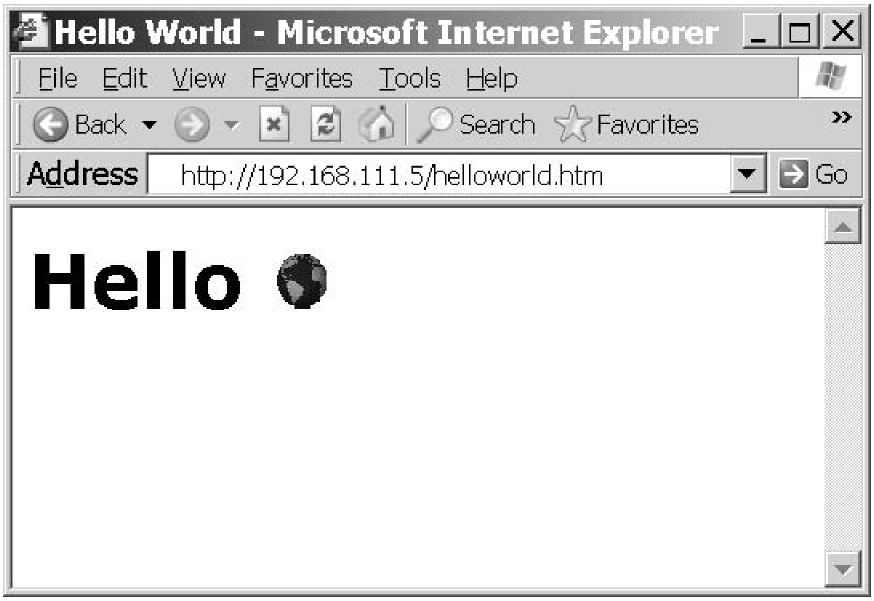

Hello World - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://192.168.111.5/helloworld.htm
Go
Hello

Figure 6-6: This basic Web page displays text and an image.

HTML elements and attributes are case-insensitive. However, elements and attributes in XHTML files must use lower case, so for XHTML compliance, use lower case for elements and attributes.

An HTML file can contain blank lines, indenting, and spaces between elements as needed for readability.

Everything between the HTML start (`<html>`) and end (`</html>`) tags is HTML-encoded text. The HTML start and end tags are optional.

The HTML-encoded text has two sections, the head and body.

The HEAD Section

The HEAD section contains information that doesn't display on the page. Everything between the `<head>` and `</head>` tags is in the HEAD section.

In the HEAD section, the `<title>` and `</title>` tags surround a title that displays in the browser window's title bar. The title also appears in the browser's Bookmarks or Favorites list if you add the page to the list.

```
<html>

<head>
  <title>Hello World</title>
</head>

<body>
  <h1> Hello <img src="earth.gif" alt="world"> </h1>
</body>

</html>
```

Listing 6-7: The HTML code for Figure 6-6's Web page.

The BODY Section

Everything between the `<body>` and `</body>` tags is in the BODY section, which contains the material that appears in the browser's main window.

Ordinary paragraph text on a Web page begins with the paragraph start tag (`<p>`). Each paragraph requires a start tag. HTML doesn't require closing paragraph tags (`</p>`), but XHTML does, so include closing tags for XHTML compliance.

Header tags provide a way to specify that paragraph text should display more prominently than ordinary text. In Figure 6-6, the word Hello is displayed as a level-1 header, enclosed by the tags `<h1>` and `</h1>`. A page can have up to six levels of headers (`<h1>` through `<h6>`). The font and size of the header text vary with the browser and how the user has configured it.

Many tags have required or optional attributes, which provide additional information to the command. The following tag tells the browser to request and display the image contained in the file *earth.gif*:

```
<img src="earth.gif" alt="world">
```

The `img` tag includes two attributes. A `src` attribute specifies the file name and the path to the file, relative to the Web site's root directory. For browsers that don't display images, the `alt` attribute specifies the text to display in place of the image.

The text that follows an attribute's equals sign is the attribute's value. In HTML, not all values need to be enclosed in quotation marks. XHTML requires quotation marks for all attribute values, so include the quotation marks for XHTML compliance.

Hyperlinks

The formatting of text and placement of images are useful in designing pages, but ultimately what made HTML and the Web popular was clickable hyperlinks to other pages. Here is an example:

```
<a href="http://www.w3.org">World Wide Web Consortium</a>
```

The `<a>` tag specifies that what follows is a link to another page. The text in quotes that follows the HREF attribute in the tag (`http://www.w3.org` in the example) names the URL to link to. The label that follows the `<a>` tag (`World Wide Web Consortium` in the example) is the text users will see on the Web page. The `</a>` tag ends the hyperlink.

The formatting that indicates a clickable link varies with the browser and how it's configured, but typically, links are underlined. With underlined links, for the above example, the browser would display only this label:

```
World Wide Web Consortium
```

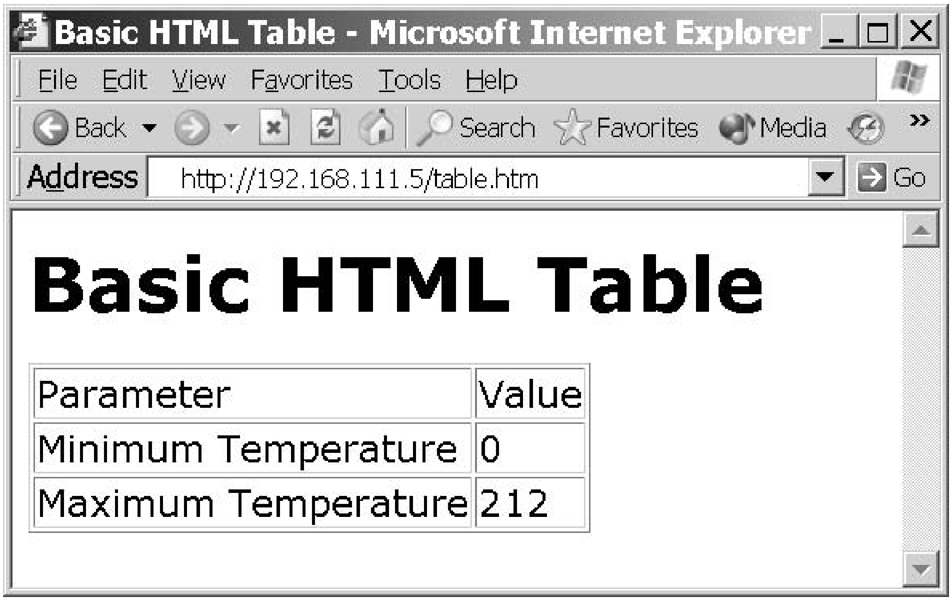
Clicking the label causes the browser to send a request for the default Web page (since no file name is specified) at the host *www.w3.org*.

Using Tables to Format Text and Images

A popular way of formatting information on Web pages is with the use of tables. An HTML table specifies the placement of cells in rows and columns. Each cell can contain page elements such as text, an image, a hyperlink, or a combination. A table makes it easy to ensure that the page elements line up as intended on the page.

Figure 6-8 shows a Web page containing basic HTML table, and Listing 6-9 is the page's HTML code. Everything between these tags:

```
<table frame="border" border="2" rules="all">
</table>
```


Basic HTML Table - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.5/table.htm Go

Basic HTML Table

Parameter	Value
Minimum Temperature	0
Maximum Temperature	212

Figure 6-8: HTML tables provide a way of formatting information on a Web page.

is part of the table. In the `table` tag, the `frame="border"` attribute specifies that the table will display a border around its perimeter. The `border="2"` attribute specifies a border width of 2 pixels. The `rules="all"` attribute specifies that the rows and columns will display rules, or lines that delineate the rows and columns in the table.

Everything between `<tr>` and `</tr>` is in a table row. Everything between `<td>` and `</td>` is data that appears in a table cell in a row. Figure 6-8's table has three rows, with two cells in each row.

The HTML specification has more about tables and the options for formatting them.

Refreshing a Page Automatically

One limitation of Web servers is that the server won't send a page unless a client requests it. For example, a server may provide a Web page with current weather information. After a client has requested the page, the page dis-

```
<HTML>

<head> <title> Basic HTML Table </title> </head>

<body>

<h1> Basic HTML Table </h1>

<table frame="border" border="2" rules="all">

<tr>
<td> Parameter </td>
<td> Value </td>
</tr>

<tr>
<td> Minimum Temperature </td>
<td> 0 </td>
</tr>

<tr>
<td> Maximum Temperature </td>
<td> 212 </td>
</tr>

</table>

</body>

</html>
```

Listing 6-9: HTML code for ‘s Web page, which displays a table.

played in the browser doesn’t update automatically if the conditions change. If a browser wants to continuously display the current conditions, it must periodically request a new, refreshed page.

You can request the latest version of a page by clicking the Refresh icon available on most browsers. It’s also possible to add code that will cause a browser to refresh the page periodically without user intervention. Placing the following line of HTML code in a Web page’s HEAD section will cause the browser to send a request for the page every 300 seconds:

```
<meta http-equiv="Refresh" Content="300">
```

The `Content` attribute specifies the number of seconds to wait before refreshing. Most browsers support this method of automatic refresh, though they may include an option to disable it.

Server Side Include Directives

A Server Side Include (SSI) directive requests a server to perform an action before serving a Web page that contains the directive. The capabilities of SSIs are limited but convenient for some applications. The Rabbit example earlier in this chapter used SSI directives to retrieve the values of variables to display on a Web page.

Server Side Includes were introduced by the Apache Group, which was formed to develop Apache HTTP Server, a popular open-source application used by many Web servers that run under UNIX, Windows, and other operating systems. Server Side Includes are now supported by many Web servers, including some embedded systems that function as Web servers. A server that supports Common Gateway Interface (CGI) programs is likely to support SSI as well. The documentation for the Apache HTTP Server includes documentation for SSI and is available from the Apache Software Foundation at *www.apache.org*.

Basics

As explained earlier in this chapter, in Web pages, SSI directives use the same delimiters as HTML comments (`<!--` and `-->`). Before serving a page that contains a directive supported by the server, the server executes the directive and replaces the delimiters and everything between them with the result of the directive. (Some directives, such as a `#config` directive that specifies formatting for other directives, have no result to display.)

Spacing is critical in SSI directives. There must be no space between the opening delimiter (`<!--`) and the directive's number sign (`#`), and there should be a space immediately preceding the closing delimiter (`-->`).

Because the server does all of the work of implementing the SSI directives, the requesting computer and its browser don't need to know anything about

SSI. The browser never sees the directives, just the results placed in the received Web pages.

Using Directives

Rabbit Semiconductor's Dynamic C supports three SSI directives: `#echo`, `#include`, and `#exec`.

#echo

The `#echo` directive inserts the current value of a variable in a requested Web page. If your server supports this directive, it's an obvious choice for displaying real-time information. To insert the value of the variable *temperature*, a Web page might contain this code:

```
<p>The temperature is <!--#echo var="temperature" --></p>
```

When the server serves the page, it retrieves the value of the temperature variable and replaces the `<!--` and `-->` delimiters and everything between them with this value. If the `temperature` variable equals 72, the paragraph appears on the page as:

```
The temperature is 72
```

The variable specified in the directive must be defined as an environment variable on the server. In Dynamic C, you define environment variables by adding an `HTTP_VARIABLE` entry for the variable in an `HttpSpec` structure, as shown in the Rabbit example earlier in this chapter. Other servers use different methods for defining environment variables.

In addition to displaying text, you can use `#echo` to display an image that reflects real-time status or conditions. In the example below, an HTML `img` tag causes the Web page to display the image contained in the file whose name is stored in the string variable `led1_image`:

```
<img src="<!--#echo var="led1_image" -->">
```

The server can set `led1_image` to different file names, such as `led_on.gif` and `led_off.gif`, depending on the current state of an LED at the server. On receiving a request for the Web page, the server inserts the current value of `led1_image`, and the Web page displays an image that matches the LED's current state.

The complement to `#echo` is `#set`, which sets the value of a variable. Dynamic C doesn't support `#set`, however.

#include

The #include directive causes the server to place the contents of the specified file in the Web page. The following `#include` directive:

```
<pre>
<!--#include file="myfile.txt" -->
</pre>
```

places the contents of *myfile.txt* in a Web page, at the location of the directive in the page. If the included file isn't HTML-encoded, precede the directive with an HTML `<pre>` tag, which tells the browser to maintain the line breaks and spacing in the file's contents. The `</pre>` tag ends the preformatted content.

#exec

The `#exec` directive can execute a command or CGI program and place the result in the Web page being served. In Dynamic C's implementation of `#exec`, the functions that the directive can execute must be specified in an `HTTPSPEC_FUNCTION` item in an `HttpSpec` structure.

The `#exec` directive can be a security risk if the system software doesn't have appropriate restrictions on what the directive can execute. For example, a Web site might display a guest book of comments from Web-site visitors. If a malicious visitor enters an `#exec` directive in the guest book, when a client requests the Web page containing the guest book, the server will parse the page for SSI directives and will attempt to execute the `#exec` directive, with possibly disastrous results.

Identifying Files that use Server Side Includes

The security issues with `#exec` directives suggest that there is good reason to limit which files a server parses for SSI directives. If the server doesn't look for directives, any directives that happen to be present won't execute and the browser will ignore them as HTML comments. Another reason to limit the files a server parses for SSI directives is to save the server from wasting time needlessly looking for directives on pages that don't have any.

The usual way to identify pages that use SSI is to give the filenames the extension *.shtml*, while plain HTML files use *.htm* or *.html*.

In Dynamic C, the `HttpType` structure specifies a handler to use with each supported file extension. In the Dynamic C example below, files with the extensions *.shtml* and *.html* each use a different handler. The *.shtml* handler parses the files for SSI directives, while the *.html* handler doesn't.

```
const HttpType http_types[] =
{
   { ".shtml", "text/html", shtml_handler},
   { ".html", "text/html", html_handler}
};
```

Other Web servers use other methods for specifying the handlers to use with different file types, but the concept is the same.

7

Serving Web Pages that Respond to User Input

Chapter 6 showed how a Web page can use HTML to display text and images, including real-time data. Many embedded Web servers also need to display pages that can respond to user input. For example, a Web page might display a virtual control panel that enables users to start, stop, or modify processes controlled by an embedded system. Or a page might display a form that enables users to enter or select values for use in configuring or controlling a device.

‘Two technologies for enabling Web pages to respond to user input are common gateway interface (CGI) programming and Java servlets. CGI programs and Java servlets can do the following:

- Retrieve the current values of variables and place them on a Web page to return to a client.
- Receive and act on data provided by a client who clicks a hyperlink or submits an HTML form.

5

Exchanging Messages Using UDP and TCP

This chapter shows how embedded systems can use the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) to send messages over a network. The messages can contain any type of data. The systems must support IP, because TCP and UDP use IP addresses to identify a message's source and destination. The In Depth section of the chapter discusses UDP and TCP in detail, including when to use each and what's involved in supporting the protocols in an embedded system.

Quick Start: Basic Communications

UDP and TCP are standard, well-supported protocols for computers that need to send and receive messages within local networks or on the Internet. Many application protocols transfer information using UDP or TCP. For example, a computer that sends a request for an IP address to a DNS server places the request in a UDP datagram. A request to a server for a Web page and the page sent in response both travel in TCP segments. But you can also use UDP and TCP to transfer messages of any type, including information in application-specific formats.

In general, UDP is a simpler protocol to implement but has no built-in support for acknowledging receipt of messages, determining the intended order of messages, or flow control. If you use a module with support for both UDP and TCP, the programming effort to use the protocols is likely to be about the same for each. In some cases, TCP programming may be easier.

This section presents examples of UDP and TCP communications. The embedded systems in the examples are the Rabbit and TINI modules introduced in Chapter 3. For both modules, the amount of programming required to exchange messages is greatly simplified because of the supporting code provided with the modules.

Before using a Rabbit or TINI module in networking applications, you need to configure the module with the networking parameters appropriate for your module and network. This chapter has information about how to configure Rabbit and TINI modules to enable communicating on a network and the Internet.

And because many embedded systems communicate with PCs, I've included some tips for VB.NET programming on a PC that communicates with embedded systems. Even if there will be no PCs in your final network design, the display, keyboard, and programming resources of a PC can make it a useful tool in the initial stages of a project.

Configuring a Device for Network Communications

As Chapter 4 explained, communications that use UDP, TCP, or other Internet protocols must use IP addresses to identify the sender and receiver of the communications (with the exception that a UDP datagram doesn't have to specify a source address). In addition, sending a message using IP may require any or all of the following: a netmask value, the IP address of a gateway, or router, and the IP address of a domain-name server. The device firmware may specify these values, or the device may request the values from a DHCP server.

Rabbit Configuration

For Rabbit modules, Dynamic C supports several ways for a network interface to obtain an IP address and related values. An application can provide the values or obtain values from a DHCP server. The program code that specifies the values or how to obtain them can be in the main application or in a macro that the main program calls. Using a macro keeps the main program free of system-specific values, makes it easy to use the same configuration in multiple programs, and enables changing a configuration by just specifying a different macro.

Dynamic C's *tcp_config.lib* file defines constants for use in static configurations and macros for implementing a variety of common configurations. You can edit the file as needed for your devices and network, or you can provide your own configuration macros in a file you create called *custom_config.lib*.

To specify a configuration macro, the program code must include the following statement:

`TCPCONFIG` *macro_number*

where `macro_number` names the configuration in a configuration file. The statement must occur before the statement `#use "dcrtcp.lib"`.

Configuring in the Application

The default macro, `TCPCONFIG 0`, does no configuring, so with this option, the main program has to provide the configuration information. `TCPCONFIG 0` is the default, so a program that uses `TCPCONFIG 0` doesn't require a `TCPCONFIG` statement at all. To use `TCPCONFIG 0` with a static IP address, a program should define values for the constants below, as needed:

```
#define _PRIMARY_STATIC_IP "192.168.111.7"
#define _PRIMARY_NETMASK   "255.255.255.0"
#define MY_NAMESERVER      "192.168.111.2"
#define MY_GATEWAY         "192.168.111.1"
```

The _PRIMARY_STATIC_IP and _PRIMARY_NETMASK strings should match the Rabbit module's IP address and the netmask of the subnet the Rabbit module resides in. If the module will access a domain name server, set MY_NAMESERVER to the IP address of the network's name server. If the module will communicate outside the local network, set MY_GATEWAY to the IP address of the subnet's gateway, or router, that connects to the outside world.

Using a Configuration File

To use static values defined in *tcp_config.lib*, use TCPCONFIG 1 in the application. The *tcp_config.lib* file defines the four constants above. Edit the statements with the values your system requires.

The following code in *tcp_config.lib* configures the interface using the values stored in _PRIMARY_STATIC_IP and _PRIMARY_NETMASK:

```
#if TCPCONFIG == 1
  #define USE_ETHERNET 1
  #define IFCONFIG_ETH0 \
    IFS_IPADDR,aton(_PRIMARY_STATIC_IP), \
    IFS_NETMASK,aton(_PRIMARY_NETMASK), \
    IFS_UP
#endif
```

The USE_ETHERNET macro is set to 1 to specify that the interface uses the system's first Ethernet port.

The IFCONFIG_ETH0 macro configures the first Ethernet port. The value of the macro is a parameter list, whose items are also macros. The IFS_IPADDR and IFS_NETMASK macros set the interface's IP address and netmask using the values defined earlier. The aton function converts strings in dotted-quad format to the binary values required by the macros. The IFS_UP macro brings up, or enables, the interface.

In a similar way, *tcp_config.lib* contains additional macros for other common configurations. For example, TCPCONFIG 3 specifies that the first Ethernet interface will obtain its IP address and other configuration values from a DHCP server. You can add your own custom configuration macros to the file as well.

TCPCONFIG values greater than 99 must be in a file called *custom_config.lib*. Create this file if you want to store custom configurations in your own file, rather than using *tcp_config.lib*.

Using ifconfig() to Set and Retrieve Network Settings

Dynamic C's ifconfig() function enables firmware to set and retrieve various network values at runtime. For example, on receiving a request to run a CGI program, a Web server might want to return a response containing a code that requests the browser to refresh the page. The response must include the URL of the page the browser should request. If the Rabbit's interface obtained its address from a DHCP server, a program can use ifconfig() to obtain the IP address to use in the URL.

Calls to ifconfig() can contain varying numbers of parameters. In the example below, the IFG_IPADDR macro returns the IP address for the default interface (IF_DEFAULT) in the variable my_ip_address. A sprintf() statement stores a URL containing the retrieved IP address in the character array redirect_to. The inet_ntoa() function converts the binary IP address returned by IFG_IPADDR to dotted-quad format.

```
longword my_ip_address;
char redirect_to[127];
char ip_dotted_quad[16];

ifconfig(IF_DEFAULT,
  IFG_IPADDR, &my_ip_address,
  IFS_END);

sprintf(redirect_to, "http://%s/index.shtml",
    inet_ntoa(ip_dotted_quad, my_ip_address));
```

Dynamic C's documentation has more details and examples for ifconfig().

Viewing Debugging Information

For debugging applications that use networking functions, the following directives are useful:

```
#define DCRTCP_DEBUG
#define DCRTCP_VERBOSE
```

DCRTCP_DEBUG enables debugging within the TCP/IP libraries. DCRTCP_VERBOSE prints debugging messages to Dynamic C's STDIO window. The global variable debug_on controls the number of messages, and thus the amount of detail revealed. To set debug_on from 0 (few messages) to 5 (maximum messages), set the third parameter in this ifconfig() statement:

```
ifconfig(IF_ANY, IFS_DEBUG, 5, IFS_END);
```

Use a lower value to decrease the number of messages and thus the amount of root memory required to display the messages. Another way to decrease the number of messages is to use directives that apply only to a specific library, such as FTP_DEBUG and FTP_VERBOSE.

When debugging is complete, remove the debugging directives and any related ifconfig() statements.

TINI Configuration

TINI modules can also configure their network interfaces using static values or values obtained from a DHCP server.

Using ipconfig

During project development, you can view and set the network configuration from within the JavaKit utility, using the command ipconfig. This ipconfig command is similar to the ipconfig utility that you can run from a command prompt under Windows.

Typing ipconfig from a command prompt in JavaKit displays information about the TINI's current network configuration:

```
TINI /> ipconfig

Hostname          : tini00e254
Current IPv4 addr.: 192.168.111.3/24 (255.255.255.0)
(active)
Current IPv6 addr.: fe80:0:0:0:260:35ff:fe00:e254/64
(active)
Default IPv4 GW   : 192.168.111.1
Ethernet Address  : 00:60:35:00:e2:54
Primary DNS       :
Secondary DNS     :
```

```
DNS Timeout        : 0 (ms)
DHCP Server        : 192.168.111.1
DHCP Enabled       : true
DHCP Lease Ends    : Tue Apr 23 08:21:48 GMT 2002
                     (23 hr, 58 min, 33 seconds left)
Mailhost           : 0.0.0.0
Restore From Flash: Not Committed
```

Typing `help ipconfig` displays the command-line options supported for performing other functions related to the network configuration:

```
TINI /> help ipconfig
ipconfig [options]

Configure or display the network settings.
 [-a IP -m mask]     Set IPv4 address and subnet mask.
 [-n domainname]     Set domain name
 [-g IP]             Set gateway address
 [-p IP]             Set primary DNS address
 [-s IP]             Set secondary DNS address
 [-t dnstimeout]     Set DNS timeout (set to 0 for
                     backoff/retry)
 [-d]                Use DHCP to lease an IP address
 [-r]               Release currently held DHCP IP address
 [-x]                Show all Interface data
 [-h mailhost]       Set mailhost
 [-C]                Commit current network configuration
                     to flash
 [-D]                Disable restoration of configuration
                     from flash
 [-f]                Don't prompt for confirmation
```

For example, to set the static IP address 192.168.111.3 and a netmask of 255.255.255.0, type:

```
ipconfig -a 192.168.111.3 -m 255.255.255.0
```

To specify that the TINI should obtain its settings from a DHCP server, type:

```
ipconfig -d
```

Saving a Network Configuration

By default, the TINI stores its network configuration values in a special area of RAM whose contents are preserved on rebooting. In the DSTINIm400 module, the RAM has battery backup, so the contents also persist after powering down. There is still a chance that the configuration data will be lost, however, either due to battery failure, or if an application calls the `blastHeapOnReboot` method, which causes the RAM's contents to be erased on the next reboot, or if a user types `reboot -a` in JavaKit, which clears the heap and system memory before rebooting.

A solution is to store the network configuration in the DSTINIm400's Flash memory, in the area reserved for this information. This area is a portion of bank 47, which is a 64-kilobyte sector that stores both network configuration data and *slush.tbin* or another application loaded into Flash memory for running on startup. To store the network configuration data in Flash memory, execute the `ipconfig -C` command in JavaKit or call the TINI's `commitNetworkState()` method. A configuration committed to Flash memory persists after boot-up, powering down, and erasing of the RAM's contents.

It's possible for the TINI to override, or ignore, a committed configuration. To do so, execute the `ipconfig -D` command in JavaKit or call the TINI's `disableNetworkRestore()` method. The TINI will then behave as if the Flash memory had no stored parameters.

To change a configuration in Flash memory, you first need to erase bank 47 in the Flash memory. The easiest way to do this is to use JavaKit to reload slush or another *.tbin* application. To load slush, open a JavaKit session with the TINI. At the JavaKit prompt, type **B0**, press **Enter**, then type **F0** and press **Enter**. This clears the TINI's RAM. From the **File** menu, select **Load File**. Browse to the location of *slush_400.tbin* and click **Open**. When the JavaKit window displays **Load complete**, the file has been loaded into the Flash memory. You now should be able to execute `ipconfig -C` again to commit new network configuration parameters to Flash memory.

Banks 40–46 in the Flash memory store critical files such as the boot loader and files that implement the runtime environment. You don't want to corrupt these files, so use caution when executing commands that write to the Flash memory.

Sending UDP Datagrams

Now that you know something about how to configure the Rabbit and TINI modules for network communications, it's time to try an application. The first example is an embedded system that periodically sends datagrams to a remote host. To make it easy to detect missing or out-of-order datagrams in this example application, each datagram contains a byte with a sequence number. The sequence number increments on each send, resetting to zero after sending 255. A second byte in the datagram is the value of a port bit on the module. A timer determines how often to send the datagrams. The first example uses a Rabbit module and the second example uses a TINI.

A UDP communication takes place between two sockets. A socket is one end of a communication path on a network. Each socket has an IP address and a port number. In a typical application, the destination is programmed to receive UDP datagrams on a specific port. As explained later in this chapter, many standard application protocols have an assigned *well-known* port. Other applications are generally free to use any port number greater than 1023.

A destination may accept datagrams from any host or only from a specific host or hosts. The destination usually doesn't care what port the source sends from.

For a Windows application that receives the datagrams sent by the Rabbit and TINI modules, see Lakeview Research's Embedded Ethernet page at *www.Lvr.com*.

Rabbit Code

The Rabbit module's Dynamic C libraries include functions and constants for use in UDP communications. Rabbit Semiconductor also provides a variety of basic example programs that show how to do common tasks such and sending and receiving data using UDP and TCP. The Rabbit example code in this chapter is adapted from Rabbit Semiconductor's examples.

In the application, the firmware specifies the IP addresses and port numbers to use and sends a datagram periodically. A real-world application could perform additional tasks when not transmitting. As a debugging aid, in various locations in the code, a `printf` statement displays status messages in the STDIO window of Dynamic C's programming environment.

Initial Defines and Declarations

The firmware uses the `TCPCONFIG 1` macro to configure the network interface to use the static IP address and netmask specified in the *tcp_config.lib* file, as described above. `LOCAL_PORT` is the port the Rabbit will use to send the datagrams. `REMOTE_IP` is the IP address of the computer the Rabbit will send datagrams to. `REMOTE_PORT` is the port on the remote computer that will receive the datagrams. In your application, you must set `REMOTE_IP` and `REMOTE_PORT` to appropriate values for your remote computer.

```
#define TCPCONFIG 1

#define LOCAL_PORT     5551
#define REMOTE_IP      "192.168.111.5"
#define REMOTE_PORT    5550
```

The `MAX_UDP_SOCKET_BUFFERS` constant sets the maximum number of socket buffers for the application. This application, which communicates with a single host, requires just one buffer:

```
#define MAX_UDP_SOCKET_BUFFERS 1
```

The `#memmap xmem` directive stores all C functions not declared as root in the extended memory area, rather than limiting storage to the 64 kilobytes of root memory.

```
#memmap xmem
```

The *dcrtcp.lib* file is the Dynamic C library that supports TCP/IP communications. Unlike other C compilers, the Dynamic C compiler doesn't use `#include` directives because its library system automatically provides the needed function prototypes and header information normally contained in included files. In place of `#include`, to enable using a library, Dynamic C requires a `#use` directive that names the file:

```
#use "dcrtcp.lib"
```

The `mysocket` variable is a Dynamic C `udp_Socket` structure that contains information about the UDP socket that will communicate with the remote host. The `sequence` variable contains the number the Rabbit will send to the remote host.

```
udp_Socket mysocket;
int sequence;
```

The main() Function

The application's `main()` function begins by calling `sock_init()` to initialize the TCP/IP stack. This call is required before calling any functions in *dcrtcp.lib*, ncluding any functions that use UDP or IP. If successful, the function returns zero. If the function doesn't return zero, the network isn't available and the program ends.

```
main()
{
  int return_value;
  sequence = 255;
  return_value = sock_init();
  if (return_value == 0) {
    printf("Network support is initialized.\n");
     }
   else {
      printf("The network is not available.\n");
      exit(0);
      }
```

A call to `udp_open()` opens the specified UDP socket, enabling communications. The call requires a pointer to the local socket (`&my_socket`), a local port number (`LOCAL_PORT`), a remote IP address (`resolve(REMOTE_IP)`) and port number (`REMOTE_PORT`) to communicate with, and either a function to call on receiving data or `NULL` to place received data in the socket's receive buffer. This application doesn't receive datagrams, so the parameter is `null`. The `resolve()` function converts an IP address string in dotted-quad format to the longword required by `udp_open()`.

If the remote IP address is zero, the socket connects to the IP address and port of the first received datagram on the socket. If the remote IP address is -1, the socket accepts datagrams from any remote host and port and sends all datagrams as broadcasts.

```
   if(!udp_open(&my_socket, LOCAL_PORT,
       resolve(REMOTE_IP), REMOTE_PORT, NULL)) {
        printf("udp_open failed.\n");
        exit(0);
        }
      else {
       printf("udp_open succeeded.\n");
      }
```

The `WrPortI()` function configures Port G, bit 6 as an output. This bit controls LED DS1 on the RCM3200 module's prototyping board. This application sends the state of bit 6 in the datagram.

```
  WrPortI(PGDDR, NULL, 0x40);
```

An endless `while` loop calls the `tcp_tick()` function and a costatement. The application must call `tcp_tick()` periodically to process network packets. Chapter 3 introduced Dynamic C's costatements. In this application, the costatement calls a routine that sends a datagram with a delay between each send. Dynamic C's `DelaySec()` function specifies the number of seconds to wait between datagrams. The application's `send_packet()` function sends the datagram.

```
   while(1) {
     tcp_tick(NULL);
     costate {
       waitfor(DelaySec(1));
       send_packet();
     }
   } // end while(1)
 } // end main()
```

Sending a Datagram

The `send_packet()` function creates and sends a datagram. The `send_buffer` array holds the data to send. For each send, the application increments the sequence number and places the number in the first byte of the `send_buffer` array. The sequence number resets to zero after sending 255.

```
 int send_packet(void)
 {
   char send_buffer[2];
   int buffer_length;
   int return_value;
   int test_bit;
```

```
sequence++;
if (sequence > 255) {
  sequence = 0;
  }
send_buffer[0] = (char)sequence;
```

The application then places the current value of Port G, bit 6 in `send_buffer`'s second byte and toggles the port bit. (The application toggles the bit only so that the value changes with each send for this example application.)

The `BitRdPortI()` function reads the bit, and `BitWrPortI()` writes to the bit. Chapter 7 has more about these functions.

```
 test_bit = (BitRdPortI(PGDR, 6));
 send_buffer[1] = (char)test_bit;

if (test_bit == 0) {
  BitWrPortI(PGDR, &PGDRShadow, 1, 6);
} else {
  BitWrPortI(PGDR, &PGDRShadow, 0, 6);
}
```

Dynamic C's `udp_send()` function sends the datagram. The function call requires a pointer to the local socket (`&my_socket`), a buffer with data to send (`send_buffer`), and the number of bytes in the buffer to send (`sizeof(send_buffer`). On success, `udp_send()` returns the number of bytes sent. On failure, the program closes and attempts to reopen the socket. If the call to `udp_open()` fails, the application ends.

```
return_value = udp_send(&my_socket, send_buffer,
    sizeof(send_buffer);

if (return_value < 0) {
 printf("Error sending datagram.  Closing and reopening
     socket.\n");
  sock_close(&my_socket);

  if(!udp_open(&my_socket, LOCAL_PORT,
      resolve(REMOTE_IP), REMOTE_PORT, NULL)) {
    printf("udp_open failed.\n");
    exit(0);
  }
}
else {
      printf("Sent: Message number %d \n", sequence);
     }
   return 1;
 } // end send_packet()
```

TINI Code

TINI users can also use UDP to communicate with remote hosts. Java's `java.net.DatagramSocket` class includes methods for sending and receiving UDP datagrams.

As a debugging aid, in various locations in the code below and the other TINI applications in this book, `System.out.println` statements write status messages to the standard output stream. If you run the TINI application from a Telnet session, the status messages display in the Telnet window.

This and some of the other example applications in this book start an endless loop that runs the application until its process is killed or a reboot. To kill a process, type `ps` at the command prompt to obtain a list of the processes currently running and the number assigned to each:

```
ps
3 processes
1: Java GC (Owner root)
2: init (Owner root)
4: UdpSend.tini (Owner root)
```

To kill a specific process, type `kill` followed by the number of the process.

```
kill 4
```

The specified process then ends.

Imports and Initial Declares

The application imports `java.io` classes to support input and output operations, `java.net` classes to support networking functions, and the TINI-specific `BitPort` class to enable reading and writing to port bits.

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

For this example, the `UdpSend` class implements the `Runnable` interface to enable the code that sends the datagrams to run in its own thread. Using a separate thread makes the code a little more complicated but also more useful because the program's main thread can perform other tasks at the same time.

The `testBit` variable is Port 5, bit 2 on the DSTINIm400's CPU. This bit controls LED1 on the module.

```
 public class UdpSend implements Runnable {

   private BitPort testBit =
         new BitPort(BitPort.Port5Bit2);
   private byte[] dataToSend;
   private DatagramPacket udpPacket;
   private DatagramSocket udpSocket;
   private int delayTime;
   private int messageCount;
   private Thread datagramSender;
   private volatile boolean sendDatagrams;
```

Starting the Thread to Send Datagrams

The class's constructor has three parameters: the IP address and port of the computer to send datagrams to (`destinationInetAddress` and `destinationPort`) and the amount of time to delay between sending datagrams (`delayTime`). A `SocketException` is thrown if the socket can't be created.

```
   public UdpSend(InetAddress destinationInetAddress,
       int destinationPort, int delayTime)
       throws SocketException {
```

The byte array `dataToSend` holds the data to send to the remote host. For this application, the datagrams are just two bytes.

```
     byte[] dataToSend = new byte[2];
```

The `delayTime` variable that the `datagramSender` thread will use is set to the value of the `delayTime` parameter.

```
     this.delayTime = delayTime;
```

Communications with the remote host use the `DatagramSocket` object `udpSocket`. The socket uses an available local port; the program code doesn't have to specify a port.

```
     udpSocket = new DatagramSocket();
```

The datagrams sent to the remote host are stored in the DatagramPacket object udpPacket. The object specifies a byte array that contains the data to send (dataToSend), the length of the byte array (dataToSend.length), and the IP address and port number to send the datagrams to (destinationInetAddress, destinationPort).

```
      udpPacket = new DatagramPacket(dataToSend,
          dataToSend.length, destinationInetAddress,
          destinationPort);
```

The datagramSender thread manages the sending of the datagrams. Using a separate thread enables the application to perform other tasks without having to wait for the thread's timer to time out. Setting the thread's setDaemon() property true creates the thread as a Daemon thread. The JVM exits when there are no user (non-Daemon) threads running. Calling the thread's start() method calls UdpSend's run() method (below).

```
      datagramSender = new Thread(this);
      datagramSender.setDaemon(true);
      datagramSender.start();
    } // end UdpSend constructor
```

The main() Method

The main() method sets the values of three variables that may change depending on the application: the delayTime value in milliseconds and the values of destinationIPAddress and destinationPort. The getByName method of InetAddress converts the IP address in dotted-quad format to the InetAddress object required by the DatagramPacket object.

```
    public static void main(String[] args)
        throws IOException {

      int delayTime = 1000;

      int destinationPort = 5550;
      String destinationIPAddress = "192.168.111.5";

       InetAddress destinationInetAddress =
          InetAddress.getByName(destinationIPAddress);
```

A call to the `UdpSend` constructor creates the `myUdpSend` object with the specified destination address, destination port, and delay time.

```
    UdpSend myUdpSend =
        new UdpSend(destinationInetAddress,
        destinationPort, delayTime);
```

A `while` loop keeps the main thread alive. In this example application, the thread spends most of its time sleeping.

```
    while(true) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        System.out.print("InterruptedException: ");
        System.out.println(e);
      }
    } // end while(true)
  } // end main
```

Stopping the Sending of Datagrams

A `stop()` method enables program code to end the `datagramSender` thread and close the socket. Otherwise, the thread ends when no user threads are running.

```
  public void stop() {
    sendDatagrams = false;
    datagramSender.interrupt();
    udpSocket.close();
  } // end stop()
```

Sending Datagrams

The `run()` method executes when the `main()` method calls the `datagram-Sender` thread's `start()` method.

```
  public void run() {
```

The `sendDatagrams` variable is initialized to true and `messageCount` is initialized to 255 so it wraps back to zero on the first message sent.

```
    sendDatagrams = true;
    int messageCount = 255;
```

A `while` loop repeatedly creates and sends a datagram, then waits `delayTime`. The loop ends when an exception or the `stop()` method sets `sendDatagrams` false.

```
      while (sendDatagrams) {
        try {
          createDatagrams();
```

`DatagramSocket`'s `send()` method sends the datagram to the IP address and port specified in `udpSocket`.

```
          udpSocket.send(udpPacket);
```

After sending a datagram, the program toggles the port bit whose value was sent in the datagram. `BitPort`'s `readLatch()` method returns the last value written to the specified bit. (The application toggles the bit only so the value changes with each send for this example application.)

```
          if (testBit.readLatch() == 0) {
            testBit.set();
          } else {
            testBit.clear();
          }
```

The thread then sleeps for the specified delay time. Calling the thread's `interrupt()` method causes an `InterruptedException`, whose catch block sets `sendDatagrams` false, ending the thread. An error when attempting to send a packet causes an `IOException`, whose catch block also sets `sendDatagrams` false, ending the thread.

```
          Thread.sleep(delayTime);

        } catch (InterruptedException e) {
          System.out.println("InterruptedException: ");
          System.out.println(e);
          sendDatagrams = false;

        } catch (IOException e) {
          System.out.print("IOException: ");
          System.out.println(e);
          sendDatagrams = false;
        }
      } // end while(sendDatagrams)
    } // end run()
```

Creating the Datagram

The `createDatagram()` method stores the datagram's two bytes in the `dataToSend` byte array. The first byte is the `messageCount` variable, which increments on each send, resetting to zero after sending 255. The second byte is the value of Port 5, bit 2. The message is stored in the byte array `dataToSend`.

```
   private void createDatagram() {
```

The message count in the datagram's first byte increments with each datagram, resetting to zero on 255.

```
    if (messageCount == 255) {
      messageCount = 0;
    } else {
      messageCount = ++messageCount;
    }

    dataToSend[0] = (byte)messageCount;
```

The datagram's second byte is the last value written to the `testBit` port bit.

```
    dataToSend[1] = (byte)testBit.readLatch();
```

The `setData()` method of `DatagramPacket` stores the byte array in the `DatagramPacket` object. The `setLength()` method trims the datagram to match the length of the data. The size of `dataToSend` sets the datagram's maximum length.

```
    udpPacket.setData(dataToSend);
    udpPacket.setLength(dataToSend.length);
    System.out.print("Message number: ");
    System.out.println(messageCount);
   } // end CreateDatagram()

} // end UdpSend
```

Receiving UDP Datagrams

The other side of UDP communications is receiving datagrams. The following applications are complements to the previous examples. A Rabbit and TINI program each wait to receive a datagram from a remote host, then display the contents of the datagram.

Rabbit Code

Much of the Rabbit code for receiving datagrams is similar to the code in the previous Rabbit example.

Initial Defines and Declarations

A `TCPCONFIG 1` macro selects a network configuration from the *tcp_config.lib* file.

The `MAX_UDP_SOCKET_BUFFERS` macro specifies the number of socket buffers to allocate for UDP sockets. This application requires one socket buffer.

`LOCAL_PORT` specifies the port that the Rabbit will receive datagrams on. Generally, any port over 1023 is acceptable. The remote host will need to know this value when sending datagrams.

`REMOTE_IP` is the IP address of the host to receive datagrams from. Set this value to the IP address of the sending host. To accept datagrams from any host, set `REMOTE_IP` to zero. To accept only broadcast packets, set `REMOTE_IP` to 255.255.255.255.

```
#define TCPCONFIG 1
#define MAX_UDP_SOCKET_BUFFERS 1
#define LOCAL_PORT 5550
#define REMOTE_IP "192.168.111.5"
```

The `#memmap` directive stores all C functions not declared as root in the extended memory area, rather than limiting storage to the 64 kilobytes of root memory.

The *dcrtcp.lib* file is the Dynamic C library that supports TCP/IP communications.

```
#memmap xmem
#use "dcrtcp.lib"
```

The `datagram_socket` variable is a Dynamic C `udpSocket` structure.

```
udp_Socket datagram_socket;
```

Receiving a Datagram

The `receive_packet()` function checks for a received datagram and if there is one, writes its contents to Dynamic C's STDIO window. The `received_data` array holds the contents of the received datagram.

```
int receive_packet()
{
  static char received_data[128];
```

The `GLOBAL_INIT` section executes only once. The `memset()` function initializes the block of memory that will hold a received datagram.

```
  #GLOBAL_INIT
  {
     memset(received_data, 0, sizeof(received_data));
  }
```

The `udp_recv()` function receives a datagram from the host specified in `datagram_socket`. The datagram is stored in `received_data`. If the return value is -1, there is no datagram and the function returns.

```
   if (-1 == udp_recv(&datagram_socket, received_data,
       sizeof(received_data))) {
    return 0;
  }
```

If there is a datagram, a `printf()` statement writes its contents to the STDIO window.

```
printf("Received bytes: %d, %d\n",received_data[0],
    received_data[1]);
return 1;
}
```

The main() Function

As in the previous Rabbit example, the `main()` function calls `sock_init()` to initialize the TCP/IP stack. If the return value isn't zero, the network isn't available.

```
main()
{
  int return_value;
  return_value = sock_init();
  if (return_value == 0) {
    printf("Nework support is initialized.\n");
```

```
      }
     else {
        printf("The network is not available.\n");
        exit(0);
        }

   printf("Opening UDP socket\n");
```

A call to `udp_open()` opens the specified UDP socket, enabling communications. The call requires a pointer to the local socket (`&datagram_socket`) and a local port number (`LOCAL_PORT`). The socket connects to the remote IP address in `resolve(REMOTE_IP)`. The fourth parameter is zero to indicate that the socket will accept datagrams from any port on the remote host. To limit the datagrams to a specific source port at the remote host, this value can instead specify a port number. The final parameter is either a function to call on receiving data or `NULL` to place received data in the socket's receive buffer. The `resolve` function converts an IP address string in dotted-quad format to the longword required by `udp_open`.

```
   if(!udp_open(&datagram_socket, LOCAL_PORT,
        resolve(REMOTE_IP), 0, NULL)) {
     printf("udp_open failed!\n");
     exit(0);
   }
```

An endless `while` loop calls the `tcp_tick()` function to process network packets and the `receive_packet()` function to check for received datagrams.

```
   while(1) {
     tcp_tick(NULL);
     receive_packet();
   }
} // end main()
```

TINI Code

The `java.net.DatagramSocket` class includes methods for receiving UDP datagrams as well as sending them. The TINI's `TINIDatagramSocket` class is a faster, memory-conserving replacement for `DatagramSocket`. In the `DatagramSocket` class in Sun's JDK, the `receive()` method allocates a new `InetAddress` object on every receive, while `TINI-`

DatagramSocket overwrites the address instead of creating a new object. This example uses TINIDatagramSocket.

Initial Imports and Declarations

In addition to the TINIDatagramSocket class, the application imports java.io classes to support input and output operations, and java.net classes to support networking functions.

```
import java.io.*;
import java.net.*;
import com.dalsemi.tininet.TINIDatagramSocket
```

The UdpReceive class implements the Runnable interface so the code that waits for and receives datagrams can run in its own thread. The program's main thread can then perform other tasks at the same time.

```
public class UdpReceive implements Runnable {

  private TINIDatagramSocket udpSocket;
  private DatagramPacket udpPacket;
  private byte[] dataReceived;
  private Thread datagramReceiver;
  private volatile boolean receiveDatagrams;
```

Starting a Thread to Receive Datagrams

In the class's constructor, the localPort parameter specifies the local port to receive datagrams on. A SocketException is thrown if the socket can't be created.

The byte array dataReceived holds the data received from a remote host. Communications with the remote host use the TINIDatagramSocket object udpSocket. The socket uses a specific local port. The sending host must send the datagrams to this port.

The datagrams received from the remote host are stored in the DatagramPacket object udpPacket. The object specifies a byte array to contain the received data (dataReceived) and the length of the byte array (dataReceived.length). For this example, the received datagrams contain just two bytes.

```
  public UdpReceive(int localPort) throws SocketException
```

```
        {
      byte[] dataReceived = new byte[2];
      udpSocket = new TINIDatagramSocket(localPort);

      udpPacket = new DatagramPacket(dataReceived,
          dataReceived.length);
```

The `datagramReceiver` thread manages the receiving of the datagrams. Using a separate thread enables the application to perform other tasks without having to wait for a datagram to arrive. Setting the thread's `setDaemon()` property true creates the thread as a Daemon thread, which ends when no user threads are running. Calling the thread's `start()` method calls `UdpReceive`'s `run()` method (below).

```
      datagramReceiver = new Thread(this);
      datagramReceiver.setDaemon(true);
      datagramReceiver.start();
    } // end UdpReceive constructor
```

The main() Method

The `main()` method sets the value of `localPort`, and a call to the `UdpReceive` constructor creates the `myUdpReceive` object with the specified port.

```
    public static void main(String[] args)
        throws IOException {

      int localPort = 5550;
      UdpReceive myUdpReceive = new
          UdpReceive(localPort);
```

A `while` loop keeps the `DatagramReceiver` thread alive. The loop spends its time sleeping. A real-world application could perform other functions here.

```
      while(true) {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          System.out.print("InterruptedException: ");
          System.out.println(e.getMessage());
        }
      } // end while(true)
    } // end main
```

Stopping the Receiving of Datagrams

A `stop()` method enables program code to end the `datagramReceiver` thread and close the socket. Otherwise, the thread ends when no user threads are running.

```
public void stop() {
  receiveDatagrams = false;
  datagramReceiver.interrupt();
  udpSocket.close();
} // end stop
```

Receiving Datagrams

The `run()` method executes when the `main()` method calls the `datagram-Receiver` thread's `start()` method.

```
public void run() {

  InetAddress senderAddress;
  receiveDatagrams = true;
```

A `while()` loop waits for datagrams until the `stop()` method sets `receiveDatagrams` false or an error occurs while receiving a datagram. The `receive()` method of `TINIDatagramSocket` returns on receiving a datagram. The `getAddress()` method returns the IP address of the sender. The `getData()` method returns a byte array with the datagram's contents.

A series of `System.out.println` statements writes the message and the sender's IP address to the console.

```
  while (receiveDatagrams) {
    try {

      System.out.println("Waiting for datagram ...");
      udpSocket.receive(udpPacket);

      senderAddress = udpPacket.getAddress();
      dataReceived= udpPacket.getData();

      System.out.println("Received message: ");
      System.out.println(dataReceived[0]);
      System.out.println(dataReceived[1]);
      System.out.println();
      System.out.print("from: ");
      System.out.println
```

```
              (senderAddress.getHostAddress());
        } catch (IOException e) {
```

If an error occurs while trying to receive a packet, `receiveDatagrams` is set to `false`, which ends the `while(receiveDatagrams)` loop and stops the thread.

```
          System.out.print("IOException: ");
          System.out.println(e.getMessage());
          receiveDatagrams = false;
        }
      } // end while(receiveDatagrams)
    } // end run()

  } // end UdpReceive
```

Exchanging Messages using TCP

With UDP, you can send a message at any time, to any computer, without first finding out if the remote computer is available to receive the message. With TCP, before exchanging data, one computer must request a connection to the other computer. The connection is between two sockets, with each socket defined by an IP address and port number.

The remote computer must respond to the request and the requesting computer must acknowledge receiving the response. When these events have occurred, a connection has been established and the computers can exchange other data. In a similar way, to close a connection, each computer sends a request to close and acknowledges the request to close received from the remote computer.

In the examples below, the embedded system's program waits for and responds to a request for a connection. When the connection has been established, the embedded system waits to receive data, reads a byte, increments it, sends it back to the remote host, and closes the connection. This code can server as a model for applications where a computer sends a request or command to an embedded system, which then returns a reply.

Rabbit Code

A Dynamic C application performs the TCP communications in the Rabbit module.

Initial Defines and Declarations

As in the Rabbit UDP example, the code begins by specifying a network configuration macro with `TCPCONFIG` and a local port for network communications. In this application, the Rabbit module accepts connection requests from any host and port, so there is no need to specify a remote IP address or port. The *dcrtcp.lib* file is the Dynamic C library that supports TCP/IP communications.

```
#define TCPCONFIG 1
#define LOCAL_PORT 5551
#memmap xmem
#use "dcrtcp.lib"

char server_buffer[255];
int bytes_read;
int return_value;
```

The `server_socket` variable is a Dynamic C `tcp_Socket` structure that specifies the socket to use for TCP communications.

```
tcp_Socket server_socket;
```

The function prototype for `service_request()` enables the `main()` function to call `service_request()` before it has been compiled.

```
void service_request();
```

The main() Function

The `main()` function begins by calling `sock_init()` to enable using TCP/IP functions. An endless `while` loop then alternates between executing the statements in a costatement that handles TCP communications and performing whatever other tasks the device is responsible for. Using a costatement for the TCP communications enables the device to do other things while waiting for a connection request or data from a remote host.

```
main() {
  int data_available;
```

```
    return_value = sock_init();
    if (return_value == 0) {
      while(1) {
```

In the costatement, Dynamic C's `tcp_listen()` function begins waiting for a connection request from a remote host to the specified local port. The call to `tcp_listen()` requires several parameters:

A pointer to a TCP socket (`&server-socket`).

The port number to listen on (`LOCAL_PORT`).

The remote computer's IP address (0 to accept requests from any IP address).

The port on the remote computer to communicate with (0 to communicate with any port).

Either a function to call when data is received or `NULL` to place received data in the socket's receive buffer (`NULL`).

Reserved parameter (`0`).

A `waitfor()` statement calls the application's `connection_established()` function. If a connection has been established with a remote host, the function returns 1 and program execution continues with the statements that follow. If there is no connection, the function returns 0 and program execution jumps to the costatement's closing brace. This gives other code in the `while` loop a chance to execute. When the code eventually loops back to the costatement, execution resumes at the `waitfor()` statement, which again calls the `connection_established()` function and continues in or exits the costatement as appropriate.

After a connection is established, a second `waitfor()` statement calls the application's `check_for_received_data()` function. The function returns 1 if there is a byte waiting to be read from the remote computer. If a byte is available or if the statement has been waiting for the number of seconds in `DelaySec()`, program execution continues with the statements that follow. Otherwise, program execution jumps to the costatement's closing brace and resumes at the `waitfor()` statement the next time through. The

DelaySec() function ensures that the waitfor statement eventually times out in case a remote host fails to send a data byte.

If a byte is available, a call to the application's service_request() function reads the byte and returns a response. This completes the communication, so the sock_close() function closes the connection and the costatement ends.

```
        costate {
          tcp_listen(&server_socket,LOCAL_PORT,0,0,NULL,0);
          printf("Waiting for connection...\n");

          waitfor (connection_established());
          printf("Connection established. \n");

          waitfor (check_for_received_data() ||
              DelaySec(20));

          data_available = check_for_received_data();
          if (data_available > 0) {
            service_request();
            }
          else {
            printf("Timeout: the remote host provided no
                data. \n");
            }
          sock_close(&server_socket);
          printf("The connection is closed. \n");
        } // end costate
```

If the call to sock_init() fails, the application ends without entering the while(1) loop.

```
        // Place code to accomplish additional tasks here.
      } // end while(1)
    } // if (return_value == 0)

    else {
      printf("The network is not available" \n");
      exit(0);
      }
  } // end main()
```

Establishing a Connection

The `connection_established` function called in the costatement checks for a connection to a remote host. Dynamic C's `sock_established()` function returns 1 if a connection has been established to the specified socket and 0 if there is no connection. This code is in a separate function to enable calling it in a `waitfor()` statement.

It's possible that a connection may be established, data received, and the connection closed before the code has a chance to call `sock_established()`. To enable reading any data received if this occurs, the function also calls Dynamic C's `sock_bytesready()` function, which returns the number of bytes read or -1 if there are no bytes.

If `sock_established` returns 1 or if `sock_bytesready()` returns a value other than -1, `connection_established()` returns 0, indicating that either the connection is established or that the connection is now closed but there is at least one byte ready to be read. Otherwise the function returns 1.

Dynamic C's `tcp_tick()` function processes network packets and must be called periodically.

```
int connection_established() {
  tcp_tick(NULL);
  if (!sock_established(&server_socket) &&
      sock_bytesready(&server_socket) == -1) {
    return 0;
  }
  else {
    return 1;
  }
} // end connection_established()
```

Checking for Received Data

The `check_for_received_data()` function called in the `main()` function's costatement finds out if there is data available to be read from an existing connection. The `sock_bytesready()` function returns the number of bytes waiting to be read or -1 if no bytes are available. The `check_for_received_data()` function returns 1 if a byte is available and 0 if there are no bytes. A call to `tcp_tick()` processes network packets.

```
int check_for_received_data() {
  tcp_tick(&server_socket);
  if (sock_bytesready(&server_socket) == -1) {
    return 0;
  }
  else {
    return 1;
  }
} // end check_for_received_data
```

Reading and Responding to Received Data

The `service_request()` function reads a received byte and returns a response to the remote host. Dynamic C's `sock_fastread()` function reads bytes from a socket into a buffer and returns the number of bytes read or -1 on error. The function requires a pointer to a socket to read from (`&server_socket`), the byte array to place the data in (`server_buffer`), and the maximum number of bytes to read (`sizeof(server_buffer)`).

```
void service_request() {
  bytes_read = sock_fastread(&server_socket,
    server_buffer, sizeof(server_buffer));
```

If a byte was received, the code increments it, resetting to 0 on receiving a byte of 255.

```
  if (bytes_read > 0) {
    printf("Byte received = %d \n", server_buffer[0]);
    if (server_buffer[0] == 255) {
      server_buffer[0] = 0;
      }
    else {
      server_buffer[0]=server_buffer[0] + 1;
      }
```

Dynamic C's `sock_write()` function writes the incremented byte to the socket, which causes the byte to be sent on the network to the remote host.

```
    return_value = sock_write(&server_socket,
        server_buffer, 1);
    if (return_value != -1) {
      printf("Byte sent = %d \n", server_buffer[0]);
      }
    else {
      printf("Error writing to socket. \n");
      }
    }  // end if (bytes_read > 0)
    else {
      printf("Error reading from socket. \n");
    }
  } // end service_request()
```

TINI Code

The TINI can also function as a TCP server that receives and responds to requests to connect and exchange data. As in the UdpSend application in this chapter, the application runs in an endless loop. A `kill` command in a Telnet session ends the application.

Imports and Initial Declares

The application imports `java.io` classes to support input and output operations, and `java.net` classes to support networking functions.

```
import java.io.*;
import java.net.*;
```

The `TcpServer` class implements the `Runnable` interface so the code that does the network communications can execute in its own thread. This leaves the main thread free to do other things.

```
public class TcpServer implements Runnable {

  private ServerSocket server;
  private int readTimeout;
  private Thread serverThread;
  private volatile boolean runServer;
```

The main() Method

The class's `main()` method sets `localPort` to the port number clients will connect to and sets `readTimeout` to the number of milliseconds the server will wait to receive data after a remote host connects. The timeout is expressed in milliseconds. The `TcpServer` object `myTcpServer` uses the `localPort` and `readTimeout` values.

```
  public static void main(String[] args)
      throws IOException {

    int localPort = 5551;
    int readTimeout = 5000;
```

```
      TcpServer myTcpServer = new TcpServer
          (localPort, readTimeout);
```

A `while` loop executes while waiting for connection requests. In this example, the thread spends its time sleeping.

```
      while (true){
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          System.out.print("InterruptedException: ");
          System.out.println(e.getMessage() );
        }
      } // end while(true)
    } // end main()
```

Initializing the Server

The constructor for the `TcpServer` class creates a thread to handle connection requests. The constructor's two parameters are the `localPort` and `readTimeout` values set in `main()`.

```
    public TcpServer(int localPort, int readTimeout)
        throws IOException {
```

A `ServerSocket` object (`server`) listens for connection requests on `localPort`, and on receiving a request, creates a socket object.

```
      server = new ServerSocket(localPort);

      System.out.print("The server is listening on port ");
      System.out.println(localPort);
```

The `readTimeout` variable used in the `run()` method is assigned the value of the `readTimeout` parameter passed to the constructor.

```
      this.readTimeout = readTimeout;
```

A separate thread (`serverThread`) handles connection requests. The thread's `setDaemon()` method is set to `true` so the server thread ends when no user threads are running. Calling the `start()` method calls the thread's `run()` routine.

```
      serverThread = new Thread(this);
      serverThread.setDaemon(true);
      serverThread.start();
```

```
} // end TcpServer constructor
```

Waiting for Connection Requests

Calling serverThread's start() method causes the thread's run() method to execute. The run() method accepts connections and calls a method to handle each connection.

A while loop executes until the runServer variable is false, which occurs on an exception or if the class's stop() method sets runServer false.

```
public void run() {
  runServer = true;

  while (runServer) {
    try {
```

On accepting a connection request, the server's accept() method creates a socket for exchanging data with the connected host. The class's handleConnection() method manages communications with the socket.

```
      Socket socket = server.accept();

      try {
        handleConnection(socket);
      } catch (IOException e) {
        System.out.print("An error occurred while
            working with a socket: ");
        System.out.println(e.getMessage());
      } finally {
        try {
```

When handleConnection() returns or if there is an exception, the routine closes the socket to release any resources used by it. If there is an exception when attempting to close the socket, no action needs to be taken. If an exception occurs while attempting to accept a connection, runServer is set to false to stop the thread.

```
          socket.close();
        } catch (IOException e) {
        }
      }
    } catch (IOException e) {
          runServer = false;
        }
      }
    } // end run()
```

Stopping the Server

The `stop()` method provides a way to stop the server under program control by setting `runServer false` and closing the socket.

```
    public void stop() {
      runServer = false;

      try {
        server.close();
      } catch (IOException e) {
      }
    } // end stop()
```

Handling a Connection

The `handleConnection()` method handles a single connection with a remote host.

```
    private void handleConnection(Socket socket)
        throws IOException {
      System.out.print("Connected to ");
      System.out.println(socket);
```

The socket timeout is set to the `readTimeout` value set in the `main()` method.

```
      socket.setSoTimeout(readTimeout);
```

An `InputStream` object reads data from the remote host, and an `OutputStream` object writes to the remote host.

```
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();
```

The `InputStream` object's `read()` method attempts to read a byte from the remote host.

If the byte is -1, the input stream is closed and no more communications can take place. Otherwise, the code increments the byte and writes the incremented value to the `OutputStream` object. To complete the communication, a call to the `OutputStream` object's `flush()` method causes the data to transmit immediately, and the output stream is closed.

```
      try {
        int b = in.read();
        if (b != -1) {
          out.write(b + 1);
          System.out.print("Writing ");
          System.out.print(b + 1);
          System.out.print(" to remote host.");
          out.flush();
          out.close();
        }
      } catch (InterruptedIOException ex) {
```

An `InterruptedIOException` indicates that the read attempt has timed out.

```
        System.out.print("The remote host sent no data
            within ");
        System.out.print(readTimeout / 1000);
        System.out.println(" seconds.");
      }
    } // end handleConnection()
  } // end TcpServer
```

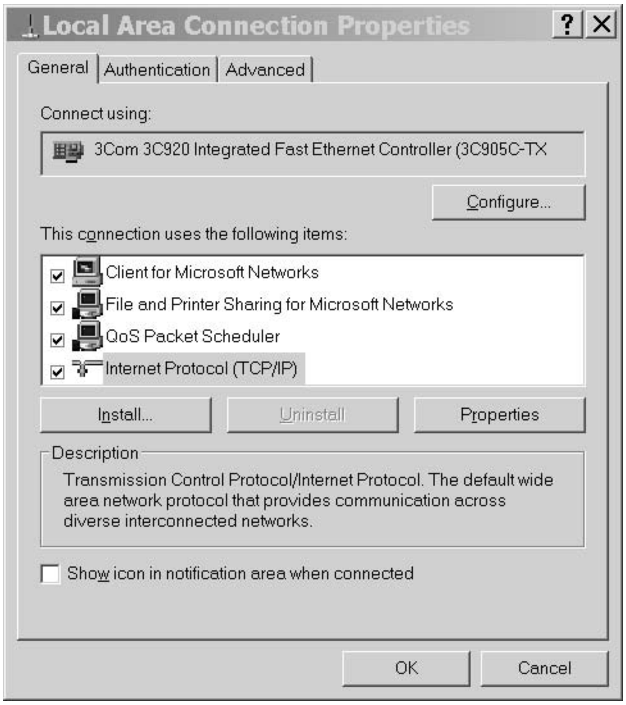
UDP and TCP from PC Applications

A PC application can communicate with any embedded system that uses UDP or TCP, including the programs above. You can write the application using any of a number of programming languages, including Visual Basic .NET and Visual C#.

About Network Programming on a PC

Windows includes plenty of support that greatly simplifies network programming and troubleshooting. Windows includes drivers that support Ethernet communications and application programming interface (API) functions that enable applications to send and receive information over a network using TCP/IP and related protocols.

For example Visual Basic .NET applications that communicate with the Rabbit and TINI programs in this chapter, see my Embedded Ethernet page at `www.Lvr.com`.



Local Area Connection Properties

General | Authentication | Advanced

Connect using:

3Com 3C920 Integrated Fast Ethernet Controller (3C905C-TX

Configure...

This connection uses the following items:

- [✓] Client for Microsoft Networks
- [✓] File and Printer Sharing for Microsoft Networks
- [✓] QoS Packet Scheduler
- [✓] Internet Protocol (TCP/IP)

Install... | Uninstall | Properties

Description

Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.

[ ] Show icon in notification area when connected

OK | Cancel

Figure 5-1: In Windows' Control Panel, Network and Connections enables installing TCP/IP support for a connection.

A custom application isn't the only way for a PC application to communicate over a network. Another option is to use a browser like Microsoft's Internet Explorer to request Web pages from computers in the network. Chapter 6 and Chapter 7 have more about how to serve Web pages from an embedded system.

To communicate over a network, a PC must have an Ethernet interface and a network connection to the embedded system the PC wants to communicate with, as described in Chapter 2.

If you access the Internet from your computer, you already have TCP/IP support installed. If you need to install TCP/IP, in Windows' Control Panel, click **Network and Connections** and right-click a connection (Figure 5-1).

If the **General** tab doesn't show Internet Protocol (TCP/IP), click the **Install** button to install it.

Using Visual Basic .NET

Network programming in Visual Basic .NET uses the `System.Net.Sockets` namespace, which includes several classes for use in network communications.

For UDP communications, at first glance, the `UdpClient` class appears to provide a convenient interface. `UdpClient` contains selected members of the `Socket` class and adds support for multicasting.

But `UdpClient` is limited in a way that makes it impractical for much beyond basic testing. `UdpClient`'s `Receive` method is synchronous, which means it blocks the program thread that calls `Receive` until a datagram arrives. If an application calls the `Receive` method and no datagram arrives, the thread that called `Receive` can do nothing but wait. You can place the `Receive` call in its own thread, leaving the main program thread free to perform other tasks while waiting, but there is no way to gracefully close a blocked thread if the data never arrives.

An alternative to `UdpClient` is the `Socket` class. By declaring a socket with `ProtocolType` set to `Udp`, you have access to all of the members of the Socket class, including the ability to use the `BeginReceive` and `EndReceive` methods in asynchronous data transfers. This means that the application doesn't have to wait for data to arrive. Instead, the program can call `BeginReceive` and continue to perform other operations. When data arrives, a callback routine runs and calls `EndReceive` to retrieve the data. And the program can close at any time.

For TCP communications, the `TcpClient` class is a little more flexible than `UdpClient`. The `SendTimeout` and `ReceiveTimeout` properties enable you to specify how long to wait for a response from a remote host before giving up. If you expect the remote host to be able to respond quickly most of the time, `TcpClient` may be suitable. Or as with UDP, you can use the `Socket` class for TCP communications.

In Depth: Inside UDP and TCP

This section explains how UDP and TCP help get data to its destination. Knowing more about how the protocols work can help in selecting which protocol to use and in using the protocol effectively. Also included is a review of options for obtaining code to support UDP, TCP, and IP in embedded systems.

The Ethernet standard specifies a way to transfer information between computers in a local network. But Ethernet alone doesn't provide some things that many data transfers require. These include naming the port, or process, that is sending the data, naming the port that will use the data at the destination, handshaking to inform the source whether the destination received the data, flow control to help data get to its destination quickly and reliably, and sequence numbering to ensure that the destination knows the correct order for messages that arrive in multiple segments. The transmission control protocol (TCP) can provide all of these. The user datagram protocol (UDP) is a simpler alternative for data transfers that only require specifying of ports or error checking. Table 5-1 compares UDP and TCP.

Figure 5-2 shows the location of UDP and TCP in a network protocol stack. UDP and TCP communicate with the IP layer and the application layer. Some applications don't require UDP or TCP, and may communicate directly with the IP layer or the Ethernet driver.

About Sockets and Ports

Every UDP and TCP communication is between two endpoints, or sockets. Each socket has a port number and an IP address.

In an Ethernet frame, the Source Address and Destination Address fields identify the sending and receiving Ethernet interfaces. A UDP or TCP communication specifies the destination more precisely by naming a port at the destination. Each TCP communication also names a source port that identi-

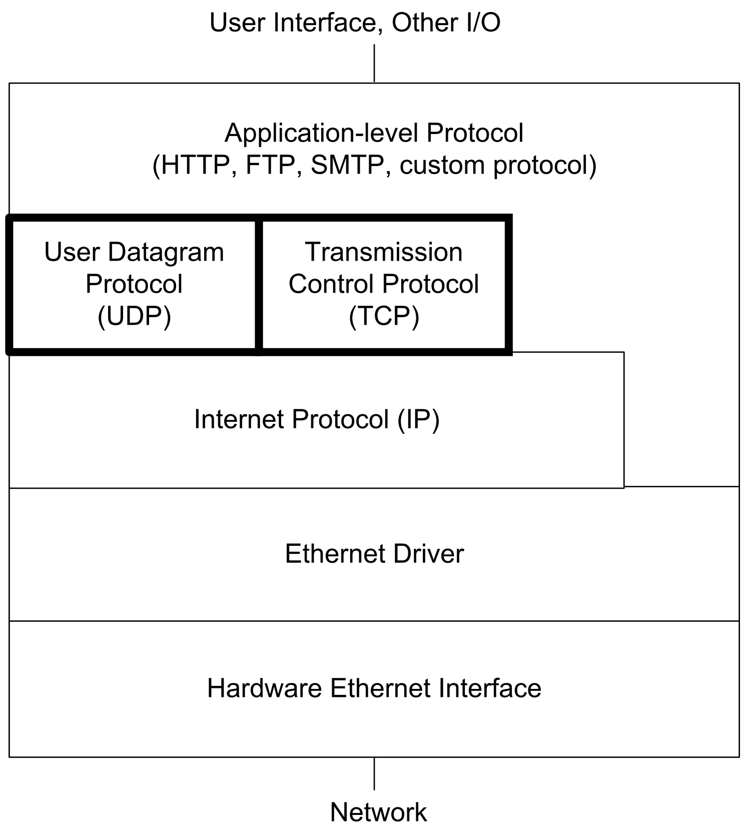

User Interface, Other I/O
Application-level Protocol
(HTTP, FTP, SMTP, custom protocol)
User Datagram Protocol (UDP)
Transmission Control Protocol (TCP)
Internet Protocol (IP)
Ethernet Driver
Hardware Ethernet Interface
Network

Figure 5-2: In the network protocol stack, the TCP and UDP layers communicate with the IP layer and the application layer. Not all communications require TCP or UDP.

fies the provider of the data being sent. Each UDP communication has a source port, but UDP datagrams aren't required to include the source-port number in the header.

A socket's port isn't a hardware port like the ports that a CPU accesses using `inp` and `out` instructions. Instead, the port number identifies the process, or task, that is providing the data being sent or using the data being received.

You can think of a socket as one end of a logical connection between computers. Unlike a physical connection, where dedicated wires and electronic components form a link, a logical connection exists only in software. Data

Table 5-1: UDP and TCP are two popular protocols for exchanging data over local networks and the Internet.

Protocol	UDP	TCP
Name of unit transmitted	datagram	segment
Source port specified to remote host?	optional	required
Must establish a connection before transferring data?	no	yes
Supports error checking?	optional	required
Supports flow control?	no	yes
Supports handshaking?	no	yes
Supports sequence numbering?	no	yes
Supports broadcasting and multicasting?	yes	no

that travels between sockets that have a logical connection doesn't have to take the same physical path every time.

The Internet Assigned Numbers Authority (IANA) (*www.iana.org*) maintains a Port Numbers list that assigns port numbers to standard processes.

There are three groups of port numbers. Values from 1 to 1023 are called well-known ports, or contact ports, and are for use by system processes or programs executed by privileged users. Table 5-2 shows examples of a few common processes and their well-known ports.

Assigning a well-known port to a process makes it easy for a computer to know what port to use when it wants to communicate with a remote computer. For example, a computer requesting a Web page normally sends the request to port 80. The receiving computer assumes that messages arriving at port 80 will use the hypertext transfer protocol (HTTP) for requesting Web pages.

Ports from 1024 to 49151 are Registered ports. An entity can request a port number from the IANA for a particular use, and the IANA maintains a list of ports it has registered. Some of the Registered ports are assigned to companies. For example, ports 5190 through 5193 are assigned to America Online. Other assignments are to processes, such as Building Automation and Control Networks (bacnet) on port 47808. Networks that don't use the

Table 5-2: Examples of standard protocols and their assigned port numbers.

Protocol	Port Number
Domain Name Service	17
FTP data	20
FTP control	21
Telnet	23
SMTP	25
Network Time Protocol	53
Gopher	70
Finger	79
HTTP	80
POP2	109
POP3	110
Quote of the Day	123

ports in this group for their registered purposes are free to use these ports for any purpose.

Ports from 49152 through 65535 are dynamic and/or private ports. The IANA doesn't assign processes to these. A network may use these ports for any purpose.

In a communication between two hosts, the values of the source and destination ports don't have to be the same and usually aren't. The source typically selects any available local port and requests to communicate with a well-known port on the destination computer. On receiving the request, the destination computer may send a reply that suggests switching the communication from the well-known port to a private port at the destination. This keeps the well-known port available to receive other new communication requests.

For communications that don't use a well-known port, such as the examples in this chapter, the computers must agree ahead of time on what ports to transmit and receive on.

UDP: Just the Basics

UDP is a basic protocol that adds only port addressing and optional error detecting to the message being sent. There is no protocol for handshaking to acknowledge received data or exchange other flow-control information. UDP is a connectionless protocol, which means that a computer can send a message using UDP without first establishing that the remote computer is on the network or that the specified destination port is available to communicate. For these reasons, UDP is also called an unreliable protocol, meaning that using UDP alone, the sender doesn't know when or if the destination received a message.

The document that defines UDP is *RFC0768: User Datagram Protocol*. It's also approved standard STD0006.

A computer that wants to send a message using UDP places the message in a UDP datagram, which consists of a UDP header followed by the data payload containing the message. As Chapter 1 explained, the sending computer places the UDP datagram in the data area of an IP datagram. In an Ethernet network, the IP datagram travels in the data field of an Ethernet frame. On receiving the Ethernet frame, the destination computer's network stack passes the data portion of the UDP datagram to the port, or process, specified in the datagram's header.

In most respects, UDP is less capable than TCP, so UDP is simpler to implement and thus more suitable for certain applications. If needed, a communication can define its own handshaking protocol for use with UDP. For example, after receiving a message, a receiving interface can send a reply containing an acknowledge code or other requested information. If the sender receives no reply in a reasonable amount of time, it can try again. But if an application needs anything more than the most basic handshaking or flow control, you should consider using TCP rather than re-inventing it for use with UDP.

UDP has one capability not available to TCP, and that is the ability to send a message to multiple destinations at once, including broadcasting to all IP addresses in a local network and multicasting to a defined group of IP addresses. Broadcasting and multicasting aren't practical with TCP because the source would need to handshake with all of the destinations.

The UDP Header and Data

The UDP header contains four fields, followed by the data being transmitted. Table 5-3 shows the fields.

Source Port Number. The source port number identifies the port, or process, on the computer that is sending the message. The source port number is optional. If the receiving process doesn't need to know what process sent the datagram, this field can be zero. The field is two bytes

Destination Port Number. The destination port number identifies the port, or process, that should receive the message at the destination. The field is two bytes.

UDP Datagram Length. The UDP datagram length is the length of the entire datagram in bytes, including the header, with a maximum of 65535 bytes. The field is two bytes.

UDP Checksum. The UDP checksum is an optional error-checking value calculated on the contents of UDP datagram and a pseudo header. The pseudo header contains the source and destination IP addresses and the protocol value from the header of the IP datagram that will contain the UDP datagram when it transmits on the network (Table 5-4). The pseudo header doesn't transmit on the network. Including the information in the pseudo header in the checksum protects the destination from mistakenly accepting datagrams that have been misrouted. The checksum value is calculated in the same way as the IP header's checksum, described in Chapter 4. The field is two bytes.

A message that travels only within a local Ethernet network doesn't need the UDP checksum because the Ethernet frame's checksum provides error checking. For a message that travels through different, possibly unknown, networks, the checksum enables the destination to detect corrupted data.

Data. A UDP datagram can be up to 65,535 bytes, and the header is eight bytes, so a datagram can carry up to 65,527 bytes of data. In practice, the

Table 5-3: A UDP header has four fields.

Field	Number of Bits	Description
Source Port Number	16	The port, or process, that is sending the datagram.
Destination Port Number	16	The port, or process, the datagram is directed to.
UDP Datagram Length	16	The datagram length in bytes.
UDP Checksum	16	Checksum value or zero.

source computer usually limits datagrams to a shorter length. One reason to use shorter datagrams is that a very large datagram might not fit in the destination's receive buffer. Or the application receiving the data may expect a message of a specific size.

Shorter datagrams may also be more efficient. When a large datagram travels through networks with different capabilities, the Internet Protocol may fragment the datagram, requiring the destination to reassemble the fragments. All of the data will still probably get to its destination, but generally it's more efficient to divide the data at the source and reassemble it at the destination, rather than relying on IP to do the work en route.

The IP standard requires hosts to accept datagrams of up to 576 bytes. An IP header with no options is 20 bytes, and the UDP header is 8 bytes. So a UDP datagram with up to 548 data bytes and no IP options should be able to reach its destination without fragmenting.

Supporting UDP in Embedded Systems

Supporting UDP in an embedded system requires the ability to add a header to data to transmit and remove the header from received data, plus support for IP.

To send a datagram using UDP, a computer in an Ethernet network must do the following:

- Place the destination port number and datagram length in the appropriate locations in the UDP header. The source port number and checksum in the header are optional. Computing the checksum requires knowing the IP addresses of the source and destination.

Table 5-4: The checksum of a UDP datagram includes the values in a pseudo header containing these five values.

Field	Size (bytes)	Source
Source Address	4	IP header
Destination Address	4	IP header
Zero	1	(none)
Protocol	1	IP header
UDP Length	2	Length in bytes of the UDP datagram including the UDP header but excluding the pseudo header

- Append the data to send to the header.
- Place the UDP datagram in the data portion of an IP datagram. The IP datagram requires source and destination IP addresses and a checksum computed on the header.
- Pass the IP datagram to the Ethernet controller's driver for sending on the network.

To receive a datagram using UDP, a computer in an Ethernet network must do the following:

- Receive an IP datagram from the Ethernet controller's driver.
- Strip the IP header from the datagram. Calculate the IP checksum and compare the result with the received value.
- If the checksums match, strip the header from the UDP datagram. If using the UDP checksum, calculate its value and compare it to the received checksum.
- Use the destination port number to decide where to pass the received data.

As the examples at the beginning of the chapter showed, if you're using a module with UDP support, the details of creating the datagrams, extracting data from a received datagram, and dealing with the checksums are handled for you. The application code just needs to provide the IP addresses, port numbers, and data to send and call a function to send the datagram, or wait to receive data in a datagram addressed to a specific port.

8

E-mail for Embedded Systems

E-mail's primary use, of course, is to enable humans to send and receive messages over a network. But many embedded systems can make good use of e-mail as well. E-mail can be a convenient way for an embedded system to exchange information with humans or even communicate with other embedded systems with no human intervention at all.

For example, a security system can be programmed to send a message when an alarm condition occurs. Or a data logger might send a message once a day with the logger's readings for the previous 24 hours. In the other direction, an embedded system might receive e-mail containing new configuration settings or other commands, requests, or data.

E-mail has a couple of advantages over other methods of communication. Recipients can retrieve and read their messages whenever they want. And if the information isn't time-critical, the sender might find it easier or more efficient to place the information in an e-mail and send it off when convenient, rather than having to respond in real time to requests for the information. Another advantage is that an account with e-mail access alone can be less expensive than an account that supports hosting a Web server or performing other TCP/IP communications.

A down side to e-mail is that recipients might not receive information as quickly as needed if they don't check their e-mail regularly or if an e-mail server at either end gets backed up and delays delivery.

This chapter begins with examples that show how a Rabbit and TINI can send and receive e-mail messages. The In Depth section has more about obtaining and using e-mail accounts for embedded systems and the protocols used to exchange e-mail on the Internet.

Quick Start: Sending and Receiving Messages

The examples that follow demonstrate how a Rabbit and TINI can send e-mail using the Simple Mail Transfer Protocol (SMTP) and receive e-mail using the Post Office Protocol 3 (POP3).

Dynamic C includes support for e-mail protocols in Rabbit modules. A TINI can send e-mail using Java's `URL` class and a protocol handler that takes care of many of the details involved in communicating with an SMTP server. For receiving e-mail, a TINI can use TCP/IP to establish a connection with a mail server's socket and exchange e-mail using the protocols supported by the server.

Sending e-mail requires the name of an SMTP server that will accept the e-mail and deliver or forward it toward its recipient. As discussed later in this chapter, the SMTP server may be at the ISP that provides the embedded system's Internet connection or at the host for the embedded system's domain name.

In a similar way, receiving e-mail requires the name of the POP3 server at the ISP or domain host that stores e-mails sent to the embedded system's mailbox. To access the mailbox, the embedded system generally must provide the account's user name and password.

The hosts of the SMTP and POP3 servers can provide the server names to use in communicating with the servers.

If the program code contains a domain name rather than an IP address for an SMTP or POP3 server, the embedded system must have a specified DNS server to request the corresponding IP address from. See Chapter 5 for more about using DNS servers with a Rabbit or TINI.

The example applications send e-mails that contain unchanging text messages and write the contents of received e-mails to the console (the STDIO window in Dynamic C or a Telnet session for the TINI). In real-world applications, the embedded system can place any kind of information in the e-mails to send and can use the information in received e-mails in any way.

Sending an E-mail from a Rabbit

Dynamic C's *smtp.lib* library contains functions that greatly simplify the code required to program a Rabbit module to send e-mail. The firmware defines strings for the sender's e-mail address, the recipient's e-mail address, the Subject line, and the message body. The `smtp_sendmail()` function then uses these values in initializing the data structures used in sending the e-mail in the format expected by the SMTP server. The `smtp_mailtick()` function handles communications with the mail server, and `smtp_status()` returns a status code when the e-mail has been sent or an error occurs.

The code that follows is an application that sends an e-mail.

Initial Defines and Declares

As explained in Chapter 5, the firmware selects a network configuration from *tcp_config.lib*.

```
#define TCPCONFIG 1
```

`SENDER` is the rabbit's e-mail address and `SMTP_SERVER` is the name of the SMTP server that will accept the e-mail and forward it toward its recipient.

You must change these values to values appropriate for your device's e-mail account and SMTP server.

```
#define SENDER      "rabbit1@Lvr.com"
#define SMTP_SERVER "mail.example.com"
```

In initiating communications with an SMTP server, the client sends a HELO command that identifies the client. By default, the Rabbit firmware sends the Rabbit's IP address as an identifier. Some mail servers require a domain name rather than an IP address. For communicating with these servers, `SMTP_DOMAIN` can set a domain name to send.

```
#define SMTP_DOMAIN "Lvr.com"
```

The `SMTP_DEBUG` macro causes all communications with the server to be displayed in the Dynamic C's STDIO window. This feature can be very helpful in debugging.

```
#define SMTP_DEBUG
```

As in the previous examples, the `#memmap` directive causes all C functions not declared as root to be stored in extended memory. The code requires the *dcrtcp.lib* library to support TCP/IP and the *smtp.lib* library for SMTP communications.

```
#memmap xmem
#use dcrtcp.lib
#use smtp.lib
```

Creating the Message

Variables hold the recipient's e-mail address, the e-mail's subject line, and the message body. The `create_message()` function sets the contents of these elements for the e-mail to be sent.

```
char recipient[64];
char subject[64];
char body[256];

void create_message() {
  strcpy(recipient, "jan@lvr.com");
  strcpy(subject, "Hello from Rabbit");
  strcpy(body, "Rabbit test message.");
}
```

Sending the Message

The `main()` routine calls the `create_message` function to compose the message then calls `sock_init()` to initialize the TCP/IP stack. The `smtp_sendmail()` function initializes internal data structures with the strings in `create_message()`. A `while` loop calls `smtp_mailtick()` repeatedly to perform communications with the SMTP server. When the server returns a value other than `SMTP_PENDING`, the `while` loop ends and the STDIO window displays the status message returned by `smtp_status()`.

```
void main()
{
  create_message();
  sock_init();

  smtp_sendmail(recipient, SENDER, subject, body);

  while(smtp_mailtick()==SMTP_PENDING)
    continue;

  switch (smtp_status())
  {
    case SMTP_SUCCESS:
     printf("The message has been sent.\n");
     break;
    case SMTP_TIME:
     printf("Timeout error. Message not sent.\n");
      break;
    case SMTP_UNEXPECTED:
     printf("Invalid response from mail server.
      Message not sent.\n");
      break;
    default:
      printf("Error. Message not sent.\n");
  }
} // end main()
```

Additional Options

The default timeout value for communications with the SMTP server is 20 seconds. The `SMTP_TIMEOUT` macro can specify a different number of seconds:

```
#define SMTP_TIMEOUT 30
```

To send a message body from a memory location instead of a string, use Dynamic C's `smtp_sendmailxmem()` function in place of `smtp_sendmail()`. Instead of a string containing the message body, the function requires the message body's starting location in memory and the message length.

Sending an E-mail from a TINI

One way to send an e-mail from a TINI is to write or obtain an SMTP client program that establishes a connection with an SMTP host and sends commands and data as needed to communicate with the host. Another option is to use the `java.net.URL` class with a protocol implementer for the URL *mailto* scheme. (See Chapter 4 for more about URL schemes.) The protocol implementer automatically handles many of the details of SMTP communications.

The TINI software supports *mailto* via `com.dalsemi.protocol.*` and `com.dalsemi.protocol.mailto.* classes`. The source code that supports *mailto* is in the file *ModulesSrc.jar*, in the *\src* directory of the TINI distribution. The following SendEmail program uses *mailto* to send an e-mail.

Imports and Initial Declares

The class imports `java.io` classes for input and output functions and `java.net` classes for networking functions. The `com.dalsemi.protocol.mailto.*` classes are required to support the `URL` class's *mailto* protocol.

The TINI's From address shouldn't change, so it's stored in the static string `MAILFROMADDRESS`. You'll need to change this value to match the address of your TINI's mailbox.

```
import java.io.*;
import java.net.*;
import com.dalsemi.protocol.mailto.*;

public class SendEmail {

  final String MAILFROMADDRESS = "tini1@Lvr.com";
```

Creating the Message

The `main()` method sets the values of three strings used in an e-mail: the recipient's e-mail address (`mailToAddress`), the Subject line (`messageSubject`), and the message body (`messageBody`). These values are passed to the `SendEmail` object `mySendEmail`.

```
public static void main(String args[])
{
  String mailToAddress = "jan@Lvr.com";
  String messageSubject = "Hello from TINI";
  String messageBody = "Test message.";

   SendEmail mySendEmail = new SendEmail(mailTo,
       subject, message);

} // end main()
```

The constructor for `SendEmail` calls the class's `send()` method to send an e-mail using the three values specified in `main()`.

```
SendEmail(String mailToAddress, String messageSubject,
    String messageBody) {
  send(mailToAddress, messageSubject, messageBody);
} // end SendEmail constructor
```

Sending the E-mail

The `send()` method does the work of sending the e-mail. The `mailURL` object is a URL object that contains the sender's and recipient's e-mail addresses in this format:

`mailto:`*mailToAddress*`?from=`*mailFromAddress*

where *mailToAddress* is the receiver's e-mail address and *mailFromAddress* is the TINI's e-mail address.

```
private void send(String mailToAddress,
    String messageSubject, String messageBody) {

  try {

    URL mailURL = new URL("mailto:" + mailToAddress +
        "?from=" + MAILFROMADDRESS);
```

The `mailConnection` object represents a connection to the SMTP server that will receive the e-mail being sent to the address in `mailTo`. The `openConnection()` method prepares to communicate with the SMTP server.

```
        Connection mailConnection =
            (Connection)mailURL.openConnection();
        mailURL.openConnection();
```

A `Printstream` object writes to the connection.

```
        PrintStream output = new
            PrintStream(mailConnection.getOutputStream());

        System.out.println("Sending the email...");
```

In sending an e-mail, the From and To headers are added automatically using the strings in `MAILFROMADDRESS` and `mailToAddress`. The application provides the Subject line, the required blank line (`\r\n`) between the end of the headers and the beginning of the message body, and the message body. The required period on a line by itself, which indicates the end of the message, is added automatically on calling the `Printstream` object's `close()` method.

```
        output.print("Subject: ");
        output.print(messageSubject);
        output.print("\r\n\r\n");
        output.print(messageBody);
        output.print("\r\n");

        output.close();
        System.out.println("The message has been sent.");
```

A `MalformedURLException` error occurs on attempting to create a URL object with incorrect URL syntax or an unsupported scheme. An `IOException` occurs on an error writing to the `PrintStream` object.

```
      } catch (MalformedURLException e) {
        System.err.print("MalformedURLException: ");
        System.err.println(e.getMessage());
      } catch (IOException e) {
        System.err.print("IOException: ");
        System.err.println(e.getMessage());
      }
    } // end send()

  } // end SendEmail
```

Adding the MAILTO Dependency to the Build

Building the SendEmail application requires a few additional considerations to enable using the *mailto* protocol handler. The build process requires `com.dalsemi.protocol` and `com.dalsemi.protocol.mailto` `classes` in *modules.jar*.

When compiling *SendEmail.java* to *SendEmail.class*, you must include the location of *modules.jar* in the bootclasspath. Here is an example command line (which you can place in a batch file) for compiling *SendEmail.java* to *Send.Email.class*:

```
javac -bootclasspath ..\..\bin\tiniclasses.jar;
     ..\..\bin\modules.jar SendEmail.java
```

When converting *SendEmail.class* to *SendEmail.tini,* use the BuildDependency utility instead of TiniConvertor. Like TiniConverter, BuildDependency converts *.class* files to *.tini* files, but BuildDependency can also specify dependencies. Here is an example command line for converting *SendEmail.class* to *Send.Email.tini*:

```
java -classpath ..\..\bin\tini.jar;%classpath%
BuildDependency -f SendEmail.class -o SendEmail.tini
-d ..\..\bin\tini.db -add MAILTO -p ..\..\bin\modules.jar
```

The `-add` option adds the `MAILTO` dependency to the project, and the `-p` option names the location of `modules.jar`.

BuildDependency is in the file *tini.jar.* To view the available options, run BuildDependency with no parameters.

Specifying the SMTP Host

To send an e-mail, you need to name an SMTP host that will receive and deliver or forward the e-mail being sent. The SendEmail application above doesn't contain this information. There are two ways to provide it. You can set the mail host in the ipconfig utility, using the `-h` option. For example:

```
ipconfig -h mail.example.com
```

Or you can provide the name in the command line that runs the program. For example:

```
java -Dmail.host=mail.example.com SendEmail.tini &
```

where *mail.example.com* is the name of the SMTP server.

A mail host specified in the command line overrides a mail host set in ipconfig.

To prevent having to type a long command line each time you run the program, create a text file that contains the command-line text, copy the file to the TINI, and run the command line by typing:

```
source filename
```

where *filename* is the name of the text file.

Receiving E-mail on a Rabbit

For retrieving e-mail from a server, Dynamic C includes the *pop3.lib* library. As with sending e-mail, the support library greatly simplifies the application code required to receive an e-mail. The following program demonstrates how a Rabbit can retrieve an e-mail. The program displays the messages in the STDIO window.

Initial Defines and Declares

As explained in Chapter 5, a `TCPCONFIG` macro selects a network configuration from *tcp_config.lib*. `POP_HOST` is the URL or IP address in dotted-quad format of the POP3 mail host for the rabbit's mailbox. `POP_USER` and `POP_PASS` are the user name and password for the rabbit's e-mail account. You must change these values to values appropriate for your system's e-mail account.

```
#define TCPCONFIG 1

#define POP_HOST  "mail.example.com"
#define POP_USER  "rabbit1"
#define POP_PASS  "embedded"
```

The `POP_PARSE_EXTRA` macro is optional, but convenient. It performs additional processing of received messages, storing the contents of the To, From, and Subject fields and the message body in separate strings.

```
#define POP_PARSE_EXTRA
```

As in the other Rabbit applications, the `#memmap` directive causes all C functions not declared as root to be stored in extended memory. The *dcrtcp.lib* library supports TCP/IP and the *pop3.lib* library supports POP3 communications.

```
#memmap xmem
#use "dcrtcp.lib"
#use "pop3.lib"

int current_message;
```

Processing and Displaying Messages

The `store_message()` function is a callback function that receives and processes downloaded messages. The function has several parameters with information about a received message. The `message_number` value is the number of the message in the series of messages being retrieved. The `to`, `from`, and `subject` strings contain the contents of the corresponding fields in an e-mail's header. The `body_line` string contains a line of text in the message body, and `body_length` is the length of `body_line`.

The function is called when headers or a line of text in a message body have been received.

```
int store_message(int message_number, char *to,
  char *from, char *subject, char *body_line,
  int body_length)
{
```

Statements in a program's `#GLOBAL_INIT` section are called only once, on program startup. In this example, the `#GLOBAL_INIT` section initializes the `current_message` variable.

```
#GLOBAL_INIT
  {
    current_message = -1;
  }
```

If the message number of a retrieved message (`message_number`) is different than the stored value in `current_message`, the function sets `current_message` equal to `message_number` and displays the message's headers in the STDIO window.

If message_number is the same as current_message, the headers have already been displayed, so there's no need to repeat them.

```
   if(current_message != message_number) {
     current_message = message_number;
     printf("MESSAGE <%d>\n", current_message);
     printf("FROM: %s\n", from);
     printf("TO: %s\n", to);
     printf("SUBJECT: %s\n", subject);
   }
```

The function writes a line of the message body to the STDIO window and returns.

```
   printf("%s\n", body_line);
   return 0;
 }
```

Retrieving Messages

The program's main() function calls sock_init() to initialize the TCP/IP stack and then calls pop3_init() to specify the callback function that will process the contents of received e-mails.

```
 void main()
 {
   static long mail_host_ip;
   static int response;

   sock_init();
   pop3_init(store_message);
```

A call to resolve(POP_HOST) returns a long value containing the IP address of the specified mail host.

```
   printf("Resolving the mail host's name...\n");
   mail_host_ip = resolve(POP_HOST);
```

The pop3_getmail() function initiates retrieving e-mail for the account specified by POP_USER and POP_PASS from the server specified in mail_host_ip. The function calls pop3_tick() repeatedly until it returns a response other than POP_PENDING to indicate that the mail retrieval is complete or has returned an error code.

```
   pop3_getmail(POP_USER, POP_PASS, mail_host_ip);
   printf("Receiving e-mail...\n\n");
   while((response = pop3_tick()) == POP_PENDING)
       continue;
```

A `switch` block displays a message that describes the result returned by `pop3_tick()`, and the program ends.

```
   switch(response)
   {
     case POP_SUCCESS:
     printf("\nThe messages have been retrieved.\n");
     break;
    case POP_TIME:
     printf("Timout error.\n");
     break;
    case POP_ERROR:
     printf("General error.\n");
     break;
    default:
      printf("Undefined error.\n");
   }
 } // end send_email
```

Additional Options

Two additional macros, `POP_DEBUG` and `POP_NODELETE`, can be useful in some situations.

For debugging, calling `POP_DEBUG` causes all communications with the POP3 server to display in Dynamic C's STDIO window.

```
 #define POP_DEBUG
```

After downloading an e-mail message, the Rabbit normally sends a POP3 DELE command to request the server to delete the message on the server. After calling `POP_NODELETE`, the Rabbit no longer sends the DELE command, and most servers will retain the messages after the Rabbit has downloaded them.

```
 #define POP_NODELETE
```

If the application doesn't require the contents of the From, To, and Subject in separate strings, don't define `POP_PARSE_EXTRA` and provide only these three parameters to the callback function: the message number, a pointer to a line of text, and the length of the text.

Receiving E-mail on a TINI

A TINI can also retrieve e-mail from a POP3 server. However, the TINI has no built-in support for POP3 communications. One option is to obtain a module with POP3 support. Or you can provide the support by writing an application that sends and responds to POP3 commands. The following application connects to a mail server, downloads any messages in the mailbox, and writes the status information and the messages to `System.out` for viewing in a Telnet session. The In Depth section of this chapter has more details about the POP3 commands the application sends.

Imports and Initial Declares

The program imports `java.net` classes for networking functions and `java.io` classes to support the input and output functions. The `java.util` package contains the `StringTokenizer` class used in reading received responses from the mail server.

The default port for POP3 servers is 110.

You must change the `USERNAME`, `PASSWORD`, and `MAILHOST` strings to match the user name, password, and the POP3 mail host for the TINI's mailbox. The mail host can be a domain name such as *mail.example.com* or an IP address in dotted-quad format.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ReceiveEmail {

  public static final int POP3PORT = 110;

  private final String USERNAME = "tini1";
  private final String PASSWORD = "ethernet";
  private final String MAILHOST = "mail.example.com";

  private BufferedReader input;
  private PrintWriter output;
  private Socket pop3Socket;

  private String mailHost;
  private String userName;
private String password;
```

The Constructor

The class's constructor uses the values passed to it to set corresponding variables.

```
public ReceiveEmail(String mailHost, String userName,
    String password) {
  this.mailHost = mailHost;
  this.userName = userName;
  this.password = password;
} // end ReceiveEmail constructor
```

Requesting Messages

The `main()` method sets the `deleteOnServer` variable and calls the `retrieveEmails()` method, which carries out the class's purpose. Set `deleteOnServer` `true` to request the mail server to delete messages after downloading, or `false` to request the server to retain the messages.

```
public static void main(String[] args) {

  boolean deleteOnServer = false;
  ReceiveEmail myReceiveEmail = new ReceiveEmail
      (MAILHOST, USERNAME, PASSWORD);
  myReceiveEmail.retrieveEmails(deleteOnServer);

} // end main()
```

The `retrieveEmails()` method calls routines to log on to the mail host, get the number of messages waiting, and read and display the messages.

The `Socket` object `pop3Socket` connects to the specified mail server's POP3 port. The Socket class's `setSoTimeout()` method enables setting a timeout in milliseconds for waiting for data from the POP3 host. A timeout causes a `java.io.InterruptedIOException`.

```
private void retrieveEmails(boolean deleteOnServer) {

  int socketTimeout = 10000;
  String response;
  try {
    System.out.print("Connecting to ");
    System.out.println(MAILHOST);
    pop3Socket = new Socket(MAILHOST, POP3PORT);
```

```
        pop3Socket.setSoTimeout(socketTimeout);
```

Reading Messages

A `BufferedReader` object reads input from the mail host, and a `PrintWriter` object writes to the mail host. `PrintWriter`'s `autoFlush` property is set to `true` to cause each `println` to automatically flush the output buffer, sending the text to the server.

On establishing a connection, the mail server returns `+OK`.

```
        input = new BufferedReader(new InputStreamReader
            (pop3Socket.getInputStream()));
        output = new PrintWriter
            (pop3Socket.getOutputStream(), true);

        response = input.readLine();
        if (response.startsWith("+OK")) {
          System.out.println("Connected to the mail host.");
```

If the connection was established, the `logOntoMailHost()` method attempts to log on. On success, the `getNumberOfMessages()` method returns the number of messages in the mailbox. If one or more messages are available, the `getMessages()` method retrieves them. The `closeConnectionWithServer()` method closes the connection with the mail host and is in a `finally` block to ensure that it executes before the method ends.

```
          if (logOntoMailHost()) {

            int numberOfMessages = getNumberOfMessages();
            if (numberOfMessages > 0) {
             getMessages(numberOfMessages, deleteOnServer);
            } else {
             System.out.println("No messages in mailbox.");
            }

          } else {
            System.out.print("Error in connecting to the
                mail host: ");
            System.out.println(response);
          }

        } // end if (response.startsWith("+OK"))
```

```
    } catch(IOException e) {
      System.err.print("IO exception: ");
      System.err.println(e.getMessage());
    } finally {
      closeConnectionWithServer();
    }
  } // end retrieveEmails
```

Logging onto the Mail Host

The `logOntoMailHost()` method uses the provided user name and password to attempt to log on to the mail host and gain access to the user's mailbox. The POP3 protocol defines `USER` and `PASS` commands for providing these values. When a command succeeds, the mail host returns `+OK`. The method returns `true` if the logon was successful and `false` if it failed.

```
  private boolean logOntoMailHost() throws IOException {

    String response;

    output.println("USER " + USERNAME);
    response = input.readLine();

    if (!(response.startsWith("+OK"))) {
      System.out.print("Password error: ");
      System.out.println(response);
      return false;
    }

    output.println("PASS " + PASSWORD);
    response = input.readLine();

    if (!(response.startsWith("+OK"))) {
      System.out.print("User name error: ");
      System.out.println(response);
      return false;
    }

      System.out.println("Logged on to the mail
          server.");
      return true;
  } // end logOntoMailHost
```

Getting the Number of Messages

The `getNumberOfMessages()` method sends a POP3 `STAT` command to retrieve the number of messages in the mailbox and the number of bytes in the messages.

The response to the `STAT` command begins with `+OK`, followed by the number of messages and the total number of bytes in the messages. The `StringTokenizer` object `st` extracts tokens, consisting of the text up to a delimiter such as a space or new-line character. The `hasMoreTokens()` method indicates whether a token is available for reading. If the first token equals `+OK`, the tokenizer extracts the tokens that follow. The method returns the number of messages or -1 on an error.

```
private int getNumberOfMessages() throws IOException {
  int numberOfMessages = 0;
  int numberOfBytes = 0;

  String response;
  output.println("STAT");
  response = input.readLine();
  System.out.println("STAT response = " + response);

  StringTokenizer st = new StringTokenizer(response);
  if (st.hasMoreTokens()) {
    if (!(st.nextToken().equals("+OK"))) {
      return -1;
    }
  }

   if (st.hasMoreTokens()) {
    numberOfMessages =
        Integer.parseInt(st.nextToken());

    if (st.hasMoreTokens()) {
      numberOfBytes =
          Integer.parseInt(st.nextToken());
    }
  }

  System.out.print("The mailbox has " );
  System.out.print(numberOfMessages);
  System.out.print(" messages in ");
  System.out.print(numberOfBytes);
  System.out.println(" bytes.");
```

```
      return numberOfMessages;
   } // end getNumberOfMessages
```

Retrieving and Displaying Messages

The `getMessages()` method retrieves the messages, displays them, and if `deleteOnServer` is `true`, requests the mail host to delete the retrieved messages when the connection closes.

```
   private void getMessages(int numberOfMessages,
       boolean deleteOnServer) throws IOException {
     String response;

     System.out.println ("Retrieving e-mail...");
```

A `for` loop steps through each message up to `numberOfMessages`, retrieving each in turn.

```
     for(int messageNumber = 1; messageNumber <=
         numberOfMessages; messageNumber++) {

       System.out.print("Retrieving message ");
       System.out.print(messageNumber);
       System.out.print(" of ");
       System.out.print(numberOfMessages);
```

The POP3 `RETR` command requests a specific message from the mail host. If the mail host's response begins with `+OK`, the `BufferedInput` object reads lines from the mail host until detecting a period on a line by itself, which indicates the end of the message.

A message body that contains a line with only a period will have an additional period added to the beginning of the line. The `startsWith()` method checks to see if the `response` string begins with a period. If it does, the `substring()` method removes the first period.

The received lines are written to the standard output stream and display in the window of a Telnet session.

```
       output.print("RETR ");
       output.println(messageNumber);

       response = input.readLine();
```

```
        if (!(response.startsWith("+OK"))) {
          System.out.print("Error reading response: ");
          System.out.println(response);
          return;
        }

        response = input.readLine();
        while(!response.equals(".")) {
          if (response.startsWith(".")) {
            response = response.substring(1);
          }
          System.out.println(response);
          response = input.readLine();
        }
```

If `deleteOnServer` is true, a POP3 `DELE` command followed by the message number requests the mail host to delete the just-retrieved message on the server.

```
        if(deleteOnServer) {
          output.print("DELE ");
          output.println(messageNumber);

          response = input.readLine();
          if (!(response.startsWith("+OK"))) {
            System.out.print("Error deleting messages: ");
            System.out.println(response);
            return;
          }
        }
      } // end for loop

      return;
    } // end getMessages()
```

Closing the Connection

The `closeConnectionWithServer()` method closes the connection with the mail server. A POP3 `QUIT` command informs that server that communications are complete, and the socket's `close()` method closes the connection.

```
    private void closeConnectionWithServer() {
      if(pop3Socket != null) {
        try {
          output.println("QUIT");
        pop3Socket.close();
        System.out.println("The connection with the mail
            server is closed.");

        } catch(IOException e) {
          System.err.print("IO exception: ");
          System.err.println(e.getMessage());
        }
    }
  } // end closeConnectionWithServer()

} // end ReceiveEmail
```

In Depth: E-mail Protocols

The examples above showed how embedded systems can use SMTP and POP3 to send and receive e-mail. This section has more about the protocols and how to use them in embedded systems.

How E-mail Works

To send and receive e-mails on the Internet, an embedded system (or any computer) must have the following:

- A connection to the Internet.
- An e-mail account with an address in the form *user_name@domain*. In the e-mail address *rabbit1@Lvr.com*, *Lvr.com* is the domain that hosts the e-mail account and *rabbit1* is the user name that identifies the owner of the account in the domain. The user also selects a password required to gain access to the account's mailbox.
- Access to incoming and outgoing mail servers. The incoming mail server accepts and stores e-mail addressed to the account and enables the user to retrieve received messages. The outgoing mail server accepts and delivers or forwards any mail the user sends.
- Support for TCP/IP and the protocols used by the mail servers in sending and retrieving e-mail. Two widely supported protocols are the Simple Mail Transfer Protocol (SMTP) for sending e-mail to a server that will

forward the e-mail toward its recipient and the Post Office Protocol Version 3 (POP3) for retrieving received e-mail from a mailbox on a server.

E-mail Accounts for Embedded Systems

An e-mail account used by an embedded system is no different from an e-mail account that anyone might use. However, in obtaining e-mail accounts, there are considerations that are specific to embedded systems.

Embedded systems tend to have limited processing power and fewer resources compared to larger computers. This means that e-mail communications should use protocols that aren't overly complex, to avoid overwhelming system resources. And second, embedded systems are likely to use their e-mail without human intervention, so they need to use protocols that enable composing, sending, retrieving, and reading messages entirely under firmware control. In other words, a Web-based e-mail account designed for users who will log onto a Web page and click through various screens to view and send messages isn't the best choice for an embedded system. An account that enables the embedded system to communicate using POP3 and SMTP commands alone is a better choice for most embedded applications.

If your embedded system will receive e-mails, you want to take special care to ensure that the e-mail address remains private. Don't give the account an easily guessed user name such as *info* or *webmaster*. And don't post the address on a Web page, because spammers will harvest the address and inundate the account with e-mails that the embedded system will have to plow through to find any valid correspondence.

Domain Hosts and ISPs

In many cases, the user or manager responsible for an account contracts with an ISP to provide everything required for Internet access, including an Internet connection, the option to set up one or more e-mail addresses, and the ability to send and receive e-mail using the ISP's mail servers. With this type of account, the ISP provides the domain name in the e-mail address and the user selects a user name that is unique to the domain.

But a user (which can be an embedded system) can also have different sources for Internet access and an e-mail account. Businesses and other entities that own a domain name often contract with a domain-hosting company for e-mail services, including the ability to create multiple e-mail addresses for the domain. Embedded systems in the domain *Lvr.com* might have the e-mail addresses *rabbit1@Lvr.com*, *rabbit2@Lvr.com*, and so on. The domain host provides a mail server that accepts and stores e-mail sent to the domain's e-mail addresses and enables the owners of the e-mail addresses to retrieve the messages on request.

ISPs generally have local connections for their customers, but a domain host doesn't have to be located physically near the computers that use the domain's e-mail accounts. To retrieve a domain's e-mail, a user may use a local ISP to gain access to the Internet and then communicate over the Internet with the domain host's mail server.

If your domain host and ISP are different entities, you need to decide which provider's mail host to use for sending e-mail. Sometimes you have a choice. In other cases, only one mail server, either at the ISP or at the domain host, will work.

The first issue in deciding what mail host to use is that the computer sending the e-mail and the mail host receiving it must support the same protocol. Embedded systems are likely to use SMTP, while some ISPs support only Web-based e-mail or other proprietary protocols.

If the sending computer's ISP doesn't have an available SMTP server to communicate with, the embedded system might be able to use a mail server at the domain host instead.

However, some domain hosts have implemented security measures that senders of e-mail need to be aware of. The security is needed because SMTP doesn't support authentication of users using passwords. A local ISP can require computers to identify themselves on connecting by providing a user name and password or a hardware identifier such as the Ethernet address of a network card or modem. The ISP can use this information to determine whether a connected computer is authorized to use the ISP's mail server.

An SMTP mail server at a domain host accessed via the Internet doesn't have information about the users who are accessing the server. Allowing anyone to use an SMTP server leaves the server open to abuse. So some hosts have implemented a type of authorization called POP-before-SMTP. This method requires a user to obtain temporary authorization to send e-mail by first checking the account for incoming e-mail. After checking for e-mail, the user is authorized to use the provider's server to send e-mail for a limited time, such as 15 minutes. After the authorization expires, the user needs to check for incoming e-mail again to regain authorization to send e-mail. If your domain host uses POP-before-SMTP authorization, your embedded system will need to comply with this protocol in order to send e-mail.

Another problem with accessing external mail servers is that some ISPs block all traffic to port 25, which is SMTP's default port, to prevent users from sending e-mail via external SMTP servers. If your ISP follows this practice and you want to use your domain host's SMTP server, check with the domain host to see if you can access their server on another port.

When you sign up for an e-mail account that uses POP3 and SMTP, the host provides the names of its incoming and outgoing mail servers. For example, the POP3 server for incoming mail might be *mail.example.com* and the SMTP server might be *smtp.example.com*. You select a user name and password, and you or the provider specifies the domain name in the e-mail address. On a PC, you can typically view the server names in your e-mail program, under **Accounts, Options**, or a similar menu item.

In the same way, an embedded system uses an account's user name, domain name, password, and servers in sending and receiving e-mail. The system's firmware can compose messages to send and parse received messages to extract the desired information.

Using the Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (SMTP) defines a reliable and efficient way of transferring e-mail to a server. Its command-and-reply protocol is basic enough to be feasible for small systems to support.

To send an e-mail, an SMTP client sends a series of commands to establish communications with an SMTP server and then sends the e-mail message for the server to deliver to its recipient or forward to another server for delivery. On receiving a command from a client, the server returns a reply code and may return a reply message or additional requested information. SMTP communications typically use TCP, but TCP isn't required.

The document that defines SMTP is RFC 2821: *Simple Mail Transfer Protocol.*

A Typical Transaction

Below is a typical session where a client establishes a connection, sends an e-mail, and closes the connection.

1. The client and server establish a TCP connection with the server's SMTP port.

   Server: `220`

2. The client identifies itself to the server.

   Client: `HELO Lvr.com`
   Server: `250`

3. The client provides the e-mail address of the sender.

   Client: `MAIL FROM <rabbit1@Lvr.com>`
   Server: `250`

4. The client provides the e-mail address of the recipient.

   Client: `RCPT TO: <jan@example.com>`
   Server: `250`

5. The client sends the e-mail's contents, including headers and ending with a period on a line by itself.

   Client: `DATA`
   Server: `354`
   Client: `From: rabbit1@Lvr.com`
   Client: `To: jan@example.com`
   Client: `Subject: Hello from Rabbit`
   Client: (blank line between e-mail header and message body)

Client: `Rabbit test message.`
Client: `.`
Server: `250`

6. The client notifies the server that it's ready to close the session.

Client: `QUIT`
Server: `221`

7. The client and server close the TCP connection.

SMTP Commands and Reply Codes

SMTP supports eleven commands for establishing communications, sending e-mail, requesting information about the server, and closing communications. Some commands have required or optional parameters. For example, with a HELO command, the client provides its domain name or IP address. After receiving a command, the server returns a 3-digit reply code. Many servers also include a text message after the reply code. For example, on receiving a QUIT command, a server at *example.com* might reply with the following reply code and text message:

```
221 example.com closing transmission channel
```

Some commands, such as HELP, request information, which the server provides following the reply code.

The SMTP standard says that the commands aren't case sensitive, but in violation of the standard, some mail servers require commands to be upper case, so using upper case is safest.

Each command and reply ends in the pair of ANSI characters 0Dh 0Ah, which is a carriage return/line feed pair, often abbreviated as CRLF. In `print` functions in program code, this pair is often expressed as `\r\n`.

The Commands

The following are the eleven SMTP commands, with an explanation and example for each:

DATA

Purpose: Announces that all of the data that follows, up to the end-of-mail indicator, is the e-mail message.
Parameters: none
Reply code on success: 354, then 250 after receiving the end-of-message indicator (a period on a line by itself).
Example:
 Client: `DATA`
 Server: `354`
 Client: `Hello,`
 Client: `This is a test message.`
 Client: `.`
 Server: `250 OK`

EHLO

Purpose: Opens communications, identifies the client, and requests information about the server. In a multi-line reply, all but the last line have a hyphen after the reply code. Some older servers support only HELO, not EHLO. Clients may use HELO, though EHLO is recommended.
Parameters: the client's domain name or IP address in dotted-quad format
Reply code on success: 250
Example:
 Client: `EHLO Lvr.com`
 Server: `250-example.com greets Lvr.com`
 Server: `250-8BITMIME`
 Server: `250-SIZE`
 Server: `250-DSN`
 Server: `250 HELP`

EXPN

Purpose: requests the server to verify that the parameter identifies a mailing list and returns the e-mail addresses of the list's members. In a multi-line reply, all but the last line have a hyphen after the reply code. Servers aren't required to support this command.
Parameter: <*the mailing list to expand*>
Reply code on success: 250 or 252
Example:
 Client: `EXPN example-list`

Server: `250-<jsmith@example1.com>`
Server: `250 <rjones@example2.com>`

HELO

Purpose: Opens communications and identifies the client.
Parameter: the client's domain name or IP address in dotted-quad format
Reply code on success: 250
Example:
Client: `HELO 192.0.2.1`
Server: `250 OK`

HELP

Purpose: Requests additional information. Servers aren't required to support this command.
Parameter: [*string that names a HELP topic*]
Reply code on success: 211 or 214
Example:
Client: `HELP`
Server: `211` *help information*

MAIL

Purpose: Initiates a transaction that sends e-mail to the server.
Parameter: `FROM:` <*sender's e-mail address*>
Reply code on success: 250
Example:
Client: `MAIL FROM: <tini1@Lvr.com>`
Server: `250 OK`

NOOP

Purpose: No operation. Verifies that the server is receiving commands.
Parameter: none
Reply code on success: 250
Example:
Client: `NOOP`
Server: `250 OK`

QUIT

Purpose: Requests the server to close the connection.

Parameter: none
Reply code on success: 221
Example:
Client: `QUIT`
Server: `221 example.com closing transmission channel`
Client and Server then close the connection.

RST

Purpose: Requests the server to cancel the current transaction and reset all buffers and state tables relating to the transaction. If the server hasn't yet acknowledged the end-of-data indicator for a message, the server discards all information relating to the message.
Parameter: none
Reply: 250
Example:
Client: `RST`
Server: `250 OK`

RCPT

Purpose: Identifies the e-mail's recipient.
Parameters: `TO:` <*sender's e-mail address*>
Reply code on success: 250 or 251
Example:
Client: `RCPT TO: <rabbit1@Lvr.com>`
Server: `250 OK`

VRFY

Purpose: Requests the server to verify that the parameter identifies the user or mailbox.
Parameter: the user's e-mail address
Reply: 250 <*user's e-mail address*>
Example:
Client: `VRFY tini1`
Server: `250 <tini1@Lvr.com>`

The Reply Codes

Table 8-1 lists the reply codes an SMTP server can return. If the reply code begins with 2, the command was successful. If the reply code begins with 3, the command was successful and the server is waiting for additional data. If the reply code begins with 5, the server didn't accept the command or carry out the requested action and the client needs to take action to correct the command before retrying.

Requirements for an SMTP Client

If your embedded system uses SMTP client code such as Dynamic C's *smtp.lib* or a *mailto* protocol handler in Java, you generally don't have to worry about the details of programming the SMTP transactions. If you're programming at a lower level, the client's program code must meet the requirements of the SMTP standard. In addition, every e-mail message must meet certain requirements.

The Client

Every SMTP client must be capable of the following:

1. The client must send the appropriate commands for establishing communications, sending e-mail, and closing communications. The minimum commands to send a message are HELO or EHLO, followed by MAIL, RCPT, DATA, and QUIT. The commands must be sent in this order.

2. The client must read received reply codes and take appropriate action, which may include retrying the command or closing the session.

3. The client must implement a timeout for receiving a reply from a command. The SMTP standard recommends timeout values ranging from 2 to 10 minutes for different operations. For example, the minimum recommended timeout for receiving a reply after sending an end-of-message indicator is 10 minutes, to allow the server time to process the message. Clients can specify other reasonable timeout values, however. If a server fails to respond and a timeout occurs, about all the client can do is close the connection and retry.

Table 8-1: An SMTP server returns one of these reply codes after receiving a command.

Reply Code	Description
211	System status or reply to HELP command.
214	Help message.
220	*domain* Service ready.
221	*domain* Service closing transmission channel. (Reply to QUIT command.)
250	Requested mail action okay and completed.
251	User is not local. Will forward to *forward path*.
252	Cannot verify user, but will accept message and attempt delivery.
354	Start the mail input.
421	*domain* Service not available, closing transmission channel.
450	Requested mail action not taken: mailbox not available (busy).
451	Requested action aborted: local error in processing.
452	Requested action not taken: insufficient system storage.
500	Syntax error, command not recognized.
501	Syntax error in parameters or arguments.
502	Command not implemented.
503	Bad sequence of commands.
504	Command parameter not implemented.
550	Requested action not taken: mailbox not available.
551	User not local; please try *forward path*.
552	Requested mail action aborted: exceeded storage allocation.
553	Requested action not taken: mailbox name not allowed (incorrect syntax).
554	Transaction failed.

4. The client must be sure that the message doesn't include a line with a period on a line by itself, which is the end-of-message indicator. If the message contains a line that begins with a period, the sender must add another period to the beginning of the line. On receiving a line of message text, an e-mail client checks to see if the line begins with a period. If it does, and if the line contains one or more additional characters, the receiver strips the period at the beginning of the line, returning the message line to its original contents.

For many embedded systems, received messages have a standard, application-specific format, and the format can be defined so that a message body never contains a period on a line by itself. In this case, the client doesn't have to worry about checking for message lines that begin with periods.

Messages

In addition to the requirements for the server, RFC standards specify requirements for e-mail messages.

The SMTP standard specifies maximum lengths that all SMTP servers must support. A user name may be up to 64 characters. A domain name may be up to 255 characters. A line in a message may be up to 1000 characters, including the two end-of-line characters. And a message may be up to 64 kilobytes. Servers must support at least these "minimum maximums," and can support larger maximums.

The document *RFC 2822: Internet Message Format* specifies the format for text messages sent as e-mail. A message consists of the following elements in order:

headers
blank line
message body

Each header field has the following format:

field_name:*field_body*\r\n

where *field_name* is the field's name (such as `From`), *field_body* is the field's contents (such as `rabbit1@Lvr.com`), and **\r\n** signifies a CRLF sequence.

The two header fields required by the specification are From and Date. From identifies the sender. Date is the date that the sender put the message into its final form. Other headers such as To and Subject are optional. Dynamic C's SMTP client automatically inserts From, To, and Subject headers using the parameters provided to the `smtp_sendmail()` and `smtp_sendmailxmem()` functions.

Not every embedded system that wants to send e-mail includes a real-time clock for obtaining Date information. A message without a Date field will reach its recipient as long as the recipient's software doesn't care that the field is missing.

The recommended format for the Date field is:

```
Date: <day> <month> <year> <time of day> <time zone>
```

or

```
Date: <day of week>, <day> <month> <year> <time of day> <time zone>
```

The month and optional day of the week are given as 3-letter abbreviations. The time is in hours and minutes since midnight. The year must use four digits. The time zone should be local.

For example,

```
Date:  11 Oct 2003 14:52 CST
```

or

```
Date: Mon, 5 Jun 2003 12:01 EST
```

For information on standard ways to send non-text information such as images or audio, see the MIME specifications in RFC 2045, 2046, and 2049.

Performance Issues

If your device has time-critical tasks to perform at the same time as it's sending e-mail, it's best to place the code that communicates with the SMTP server in its own thread or task, so the CPU can do other things while waiting for the server to respond.

Sending E-mail with a URL

Another option for sending e-mail is to use a URL with the *mailto* scheme. Chapter 4 introduced URLs and schemes such as *http*, *ftp*, and *mailto*. The scheme identifies the protocol that a browser or other software will use in sending the request specified in the URL.

When you click a typical *mailto* link on a Web page, the browser creates a new e-mail message in the PC's default e-mail program and fills in the To:

header with the `mailto` address. A user can then compose and send a message to that address.

As the TINI example in this chapter showed, embedded systems can use the *mailto* protocol to send e-mail messages created in firmware. In Java, the `URL` class represents a URL, and a protocol implementer for the *mailto* scheme handles the details of communicating with an SMTP server. A basic URL that uses the *mailto* scheme has the following form:

```
mailto:tini1@Lvr.com
```

RFC 2368: The mailto URL scheme extends the *mailto* URL scheme defined in RFC1738. Under RFC2368, a *mailto* URL can also contain one or more headers and even the message body. For example, to include a From: address, use this format:

```
mailto:jan@Lvr.com?from=tini1@example.com
```

A question mark separates the recipient's e-mail address and the From header.

Use `&` to concatenate additional headers and the message body. For example:

```
mailto:jan@Lvr.com?from=tini1@example.com&
    subject=greeting&body=hello%20from%20TINI!
```

Characters that are reserved in HTML and in this URL scheme must be encoded. Encode a space (as in the message body above) as `%20`. Encode a question mark (?) as `%3`, an ampersand (&) as `&amp`, a percent sign (%) as `%25`, and a line break in the message body as `%0D%0A`.

Using the Post Office Protocol

SMTP enables a computer to send e-mail. A complementary protocol is the Post Office Protocol Version 3 (POP3), which enables a computer to download e-mail from a server.

The standard that defines POP3 is *RFC 1939: Post Office Protocol - Version 3*, the third edition of the protocol first described in RFC 918.

As with SMTP, in a POP3 communication, a client sends a series of commands to a server and the server returns a response to each command. POP3 communications travel in TCP segments. The default port for POP3 communications is 110.

The POP3 standard defines twelve commands. Some commands have required or optional parameters that follow the command.

Every POP3 response begins with a status indicator: +OK on success and -ERR to respond to a command that is not recognized, not implemented, or has incorrect syntax. For some commands, requested information may follow the status indicator, and a server may provide additional text to explain the status. The status indicators are always upper case.

The POP3 standard defines three states in a session. After a TCP connection has been established and the server has sent a greeting, the session is in the Authorization state. After the client identifies itself and the server has acquired resources associated with the client's mailbox, the session is in the Transaction state. In this state, the client has exclusive access to the mailbox and the client can request services from the server. After the client has sent a QUIT command, the session is in the Update state. The server releases resources associated with the client, deletes messages marked for deleting, and returns a response. The client and server then close the TCP connection.

A newer and more flexible alternative to POP3 is the Interactive Mail Access Protocol (IMAP) defined in RFC 1730. IMAP enables a user to select messages to download, move files among multiple mailboxes on the server, and share a mailbox with other clients. IMAP also has more efficient handling of MIME attachments. For the needs of a typical embedded system, however, POP3's capabilities are sufficient and easier to implement.

A Typical POP3 Transaction

Below is a typical session where a client establishes a connection, retrieves an e-mail, and closes the connection.

1. The client and server establish a TCP connection with the server's SMTP port.

Server: `+OK`

2. The client sends a user name.

   Client: `USER tini1`
   Server: `+OK`

3. The client sends a password.

   Client: `PASS ethernet`
   Server: `+OK`

4. The client requests a listing of the number of messages in the mailbox and the total number of bytes in the messages.

   Client: `STAT`
   Server: `+OK 1 856`

5. The client requests to retrieve message 1.

   Client: `RETR 1`
   Server: `+OK`
   Server: *the message contents*
   Server: .

6. The client notifies the server that it's ready to close the session.

   Client: `QUIT`
   Server: `+OK`

7. The client and server close the TCP connection.

POP3 Commands

The following are POP3's twelve commands. The commands are case-insensitive. All servers must support seven of the commands and the rest are optional, as noted.

APOP

Purpose: requests user authentication using a method that doesn't require transmitting an unencrypted password. To obtain the required MD5-digest-string parameter, the client applies the MD5 algorithm described in RFC 1321 to the timestamp in the server's greeting and a secret string shared by the client and server. For a specific mailbox, a server generally supports either PASS or APOP.
Servers required to support: no

Parameters: *user_name <MD5_digest_string>*
Reply on success: `+OK`
Example:
Client: `APOP jan` *<16-byte string in hexadecimal format>*
Server: `+OK`

DELE

Purpose: requests to mark a message for deleting. Normally, users will want to delete retrieved messages to prevent filling the mailbox and retrieving the same messages over and over. A server may also delete retrieved messages automatically, even if the client doesn't send a DELE command, or a server may delete messages that have been retrieved but not deleted after a specified time limit.
Servers required to support: yes
Parameters: *message_number*
Reply on success: `+OK`
Example:
Client: `DELE 4`
Server: `+OK message 4 deleted`

LIST

Purpose: requests a scan listing containing the number of bytes in the requested message or all messages if no message number is specified.
Servers required to support: yes
Parameters: [*message_number*]
Reply on success: `+OK` *number_of_message number_of_bytes*
If there are multiple messages, the server returns a multi-line reply.
Example:
Client: `LIST 2`
Server: `+OK 2 130`

NOOP

Purpose: no operation. Indicates that the connection to the server is valid.
Servers required to support: yes
Parameters: none
Reply on success: `+OK`
Example:
Client: `NOOP`

Server: `+OK`

PASS

Purpose: Provides a password for authentication. For a specific mailbox, a server generally supports either PASS or APOP.
Servers required to support: yes
Parameters: *password*
Reply on success: `+OK`
Example:
Client: `PASS embedded`
Server: `+OK`

QUIT

Purpose: requests the server to delete all messages marked for deleting and close the connection.
Servers required to support: yes
Parameters: none
Reply on success: `+OK`
Example:
Client: `QUIT`
Server: `+OK`

RSET

Purpose: unmark any messages marked for deleting
Servers required to support: yes
Parameters: none
Reply on success: `+OK`
Example:
Client: `RESET`
Server: `+OK`

RETR

Purpose: requests a message.
Servers required to support: yes
Parameters: *message_number*
Reply on success: `+OK` *number_of_bytes* followed by the message and ending in a period on a line by itself
Example:

Client: `RETR 5`
Server: `+OK 212 octets`
message contents
.

STAT

Purpose: requests a drop listing containing the number of messages in the mailbox and the total number of bytes in the messages.
Servers required to support: yes
Parameters: none
Reply on success: `+OK` *number_of_messages number_of_byte*s
Example:
Client: `STAT`
Server: `+OK 2 508`

TOP

Purpose: requests a message's headers plus the specified number of the message's top lines.
Servers required to support: no
Parameters: *message_number number_of_lines_to_receive*
Reply on success: `+OK`
Example:
Client: `TOP 2 3`
Server: `+OK`
Server: `From: tini2@Lvr.com`
Server: `To: controlcenter@Lvr.com`
Server: `Subject: Status Report`
Server: `Date: 28 Jul 2003 10:21 CST`
Server: `Subject: HighTemp=101`
Server: `Subject: LowTemp=13`
Server: `Subject: MedianTemp=56`
Server: .

UIDL

Purpose: requests a unique-id listing for one or all messages. The unique id is a string specified by the server and consisting of one to 70 characters in the range 21h to 7Eh. The value identifies a message in the user's mailbox and persists across sessions.

Servers required to support: no
Parameters: [*message_number*]
Reply on success: `+OK` *message_number unique_id*
If there is no message number, the server returns a multi-line reply with information about each message in turn.
Example:
Client: `UIDL 3`
Server: `+OK 3` *unique-id for message 3*

USER

Purpose: provides a user name for authentication.
Servers required to support: no
Parameters: *user_name*
Reply on success: `+OK`
Example:
Client: `USER tini1`
Server: `+OK`

Requirements for a Client

Every POP3 client must be capable of the following:

1. The client must send the appropriate commands for establishing communications, retrieving e-mail, and closing communications. The minimum commands to check for mail and retrieve messages from a mailbox protected with a user name and password are USER, PASS, STAT, RETR, and QUIT. The commands must be sent in this order.

2. The client must read received replies and take appropriate action on receiving an -ERR reply.

3. The POP3 standard doesn't talk about timeouts, but a client application will probably want to time out and close the connection if the server fails to respond to a command within a reasonable time.

4. A line of message text that begins with a period transmits with an extra period at the beginning to prevent the line from appearing as an end-of-message indicator. In receiving a line that begins with a period, the client should check to see if the line contains one or more additional charac-

ters. If it does, the client should strip the first period from the line and consider the line part of the message, not the end-of-message indicator.

Some embedded application can define a standard, application-specific format that doesn't allow a period on a line by itself in received messages. In this case, the client doesn't have to worry about checking for message lines that begin with periods.

Messages

As explained earlier in this chapter, RFC standards specify requirements for e-mail messages. The receiver of an e-mail can use the standard header fields to filter messages by sender or subject.

Performance Issues

If your device has time-critical tasks to perform at the same time it's receiving e-mail, it's best to place the code that communicates with the POP3 server in its own thread or task so the CPU can do other things while waiting for the server to respond.

9

Using the File Transfer Protocol

The previous chapters have shown several ways that an embedded system can send and receive information on networks. The options have included applications that send messages using UDP and TCP, Web pages with dynamic content, and e-mail. Another possibility that some systems can find useful is the File Transfer Protocol (FTP), which defines a way for computers to send and receive information stored in files.

This chapter includes examples that show how the Rabbit and TINI can function as FTP servers and FTP clients, followed by details about FTP and its capabilities.

9

Using the File Transfer Protocol

The previous chapters have shown several ways that an embedded system can send and receive information on networks. The options have included applications that send messages using UDP and TCP, Web pages with dynamic content, and e-mail. Another possibility that some systems can find useful is the File Transfer Protocol (FTP), which defines a way for computers to send and receive information stored in files.

This chapter includes examples that show how the Rabbit and TINI can function as FTP servers and FTP clients, followed by details about FTP and its capabilities.

Quick Start: FTP Clients and Servers

The Rabbit and TINI modules each include FTP support that helps in using a module as an FTP client or server. For the Rabbit, Dynamic C's *ftp.lib* and *ftp_server.lib* libraries provide functions for transferring information in files. For the TINI, support is available in the `URL` and `URLConnection` classes and in the TINI's `FTPClient` and `FTPserver` classes.

A Rabbit or TINI client application can communicate with just about any FTP server, in a local network or on the Internet. And you can use just about any standard FTP client application or a command-line interface to access files hosted by a Rabbit or TINI FTP server.

The example applications send text files and write the contents of received files to the console (the STDIO window in Dynamic C or a Telnet session for the TINI). In real-world applications, the embedded system can place any kind of information in the files to send and can use the information in received files in any way.

Rabbit FTP Client

The following examples show how a Rabbit can exchange files with an FTP server. A Dynamic C application can use one of two sources for files to send and receive. Many basic applications can store the files in buffers in root memory. For transferring large amounts of data, for generating a file's contents on request, or for processing received data on receipt, a data-handler callback function can receive requested files or generate files to send.

Retrieving a File

This example shows how a Rabbit module can retrieve a file, store its contents in a buffer, and write the contents of the file to Dynamic C's STDIO window.

Initial Defines and Declares

As explained in Chapter 5, a TCPCONFIG macro selects a network configuration.

```
#define TCPCONFIG 1
```

Various parameters enable communicating with a specific FTP server. You must change REMOTE_HOST, REMOTE_USERNAME, REMOTE_PASSWORD, REMOTE_FILE, and REMOTE_DIR to values appropriate for the FTP server your Rabbit will communicate with.

REMOTE_HOST is the domain name or IP address of the remote FTP server. REMOTE_PORT is the port on the FTP server to connect to. Set this value to zero to connect to the default port for the FTP control connection (21). REMOTE_USERNAME and REMOTE_PASSWORD are the user name and password that enable access to a user area on the FTP server.

```
#define REMOTE_HOST     "ftp.example.com"
#define REMOTE_PORT     0
#define REMOTE_USERNAME "embedded"
#define REMOTE_PASSWORD "ethernet"
```

Additional values specify the directory to change to on connecting to the FTP server (REMOTE_DIR) and the name of the file the Rabbit will retrieve (REMOTE_FILE). Set REMOTE_DIR to "/" to specify the server's root directory.

```
#define REMOTE_DIR      "/usr/embedded/"
#define REMOTE_FILE     "testfile.txt
```

If USE_PASSIVE is defined, PASSIVE_FLAG is set to FTP_MODE_PASSIVE, which causes the Rabbit to request to use FTP's passive mode in opening the data channel for file transfers. Passive mode can be useful when communicating through a firewall. The In Depth section of this chapter has more on passive mode.

```
#define USE_PASSIVE

#ifdef USE_PASSIVE
  #define PASSIVE_FLAG  FTP_MODE_PASSIVE
#else
  #define PASSIVE_FLAG  0
#endif
```

The `#memmap xmem` directive causes all C functions not declared as root to be stored in extended memory. The *dcrtcp.lib* library supports TCP/IP, and *ftp_client.lib* supports FTP client communications.

```
#memmap xmem
#use "dcrtcp.lib"
#use "ftp_client.lib"
```

The `file_buffer` array holds the retrieved file and should be large enough to hold any file being requested.

```
char file_buffer[2048];
```

The main() Routine

The `main()` routine begins by calling `sock_init()` to initialize the TCP/IP stack. The `retrieve_file()` function then requests a file from the remote FTP server. If `retrieve_file()` fails, it returns 1 and the program ends with an error code of 1. On success, the `main()` routine returns zero.

```
int main()
{
  int return_value;
  return_value = sock_init();
  if (return_value == 0) {
    printf("Network support is initialized.\n");
    }
  else {
    printf("The network is not available.\n");
    exit(2);
  }

  if (retrieve_file()) {
    exit(1);
  }
  return 0;
} // end main()
```

Requesting a File

The `retrieve_file()` function requests the file and returns zero on success.

```
int retrieve_file()
{
   longword file_size;
   int byte_in_file;
   int return_value;

   printf("Preparing to download %s...\n", REMOTE_FILE);
```

The `ftp_client_setup()` function initiates the request for the file. Nine parameters provide the information required to request the transfer. The `REMOTE_HOST`, `REMOTE_USERNAME`, `REMOTE_PASSWORD`, `REMOTE_FILE`, `REMOTE_DIR`, and `file_buffer` parameters are defined above.

The `FTP_MODE_DOWNLOAD` constant specifies that the Rabbit wants to retrieve (rather than send) a file. A logical OR of this value with `PASSIVE_FLAG` causes the Rabbit to request to use passive mode if `USE_PASSIVE` was defined earlier. The `sizeof(file_buffer)` parameter is the length of the buffer that will contain the retrieved file.

```
   return_value = ftp_client_setup(
     resolve(REMOTE_HOST),
     REMOTE_PORT,
     REMOTE_USERNAME,
     REMOTE_PASSWORD,
     FTP_MODE_DOWNLOAD|PASSIVE_FLAG,
     REMOTE_FILE,
     REMOTE_DIR,
     file_buffer,
     sizeof(file_buffer));
```

The function returns zero on success. The function fails if the host address is zero, if `sizeof(file_buffer)` is negative, or if there are no available socket buffers to open an internal control socket to the FTP server. If the function fails, the program ends with an exit code of 1.

```
  if (return_value != 0) {
     printf("FTP setup failed.\n");
     exit(1);
  }
```

The `ftp_client_tick()` function manages communications with the FTP server. The function returns zero while pending, 1 on success, and a value from 2 to 6 to indicate an error. The program loops until the function returns a non-zero value.

```
   printf("Looping on ftp_client_tick()...\n");
   while( 0 == (return_value = ftp_client_tick()) );
```

On success, a call to the ftp_client_xfer() function returns the size of the file retrieved. Dynamic C's STDIO window displays the file size and the contents of the file.

On failure, a printf() statement displays an error message. A call to ftp_last_code() returns the most recent message code returned by the FTP server.

```
   if( 1 == return_value ) {
     file_size = ftp_client_xfer();
     printf("The file has been received.
       File size: %d bytes.\n", file_size);

     printf("Contents of file:\n");
     for (byte_in_file = 0; byte_in_file <=
         (file_size - 1); byte_in_file++)
     printf("%c",file_buffer[byte_in_file]);
     printf("\n");
     return 0;
   } else {
     printf("FTP download failed: status = %d, last code =
         %d\n", return_value, ftp_last_code());
     return 1;
   }
 } // end retrieve_file
```

Sending a File

In a similar way, a Rabbit can use the *ftp.lib* library to send a file to an FTP server.

Initial Defines and Declares

Much of the program code is similar to the previous example, including these initial statements that provide system-specific and application-specific information for the transfer and name the libraries the program uses. You must change REMOTE_HOST, REMOTE_USERNAME, REMOTE_PASSWORD and REMOTE_DIR to values appropriate for the FTP server your Rabbit will communicate with.

```
#define TCPCONFIG 1
#define REMOTE_HOST     "ftp.example.com"
#define REMOTE_PORT     0
#define REMOTE_USERNAME "embedded"
```

```
#define REMOTE_PASSWORD "ethernet"
#define REMOTE_DIR      "/usr/embedded/"
#define REMOTE_FILE     "testfile.txt"
#define USE_PASSIVE

#ifdef USE_PASSIVE
  #define PASSIVE_FLAG  FTP_MODE_PASSIVE
#else
  #define PASSIVE_FLAG  0
#endif

#memmap xmem
#use "dcrtcp.lib"
#use "ftp_client.lib"
```

The `file_buffer` array holds the data that the Rabbit will transfer in a file. This example uses a small 10-byte buffer.

```
char file_buffer[10];
```

The main() Function

The `main()` function calls `create_file()` to place data in the array that will be sent as a file to the FTP server. A call to `sock_init()` initializes the TCP/IP stack. If the initialization fails, the program ends with an error code of 2. The `send_file()` function sends the file. If the attempt to send the file fails, the program ends with an error code of 1. On success, the program returns zero.

```
int main()
{
  int return_value;
  create_file();

  return_value = sock_init();
  if (return_value == 0) {
    printf("Network support is initialized.\n");
    }
  else {
    printf("The network is not available.\n");
    exit(2);
  }
  if (send_file())
    exit(1);
  return 0;
} // end main()
```

Creating the File

For this example, the data in the file is the string `"test data"`, terminating in a null character (`\0`). Of course, a file can contain any text or binary data.

```
create_file(void) {
  file_buffer[0]='t';
  file_buffer[1]='e';
  file_buffer[2]='s';
  file_buffer[3]='t';
  file_buffer[4]=' ';
  file_buffer[5]='d';
  file_buffer[6]='a';
  file_buffer[7]='t';
  file_buffer[8]='a';
  file_buffer[9]='\0';
} // end create_file
```

Sending the File

As in the `retrieve_file()` function in the previous example, the `send_file()` function calls `ftp_client_setup()`, followed by `ftp_client_tick()`.

```
int send_file(void)
{
  int return_value;
  printf("Calling ftp_client_setup() to upload %s...\n",
      REMOTE_FILE);
```

The parameters for `ftp_client_setup()` are the same as in the previous example except for the last value, which contains the size of the file being transmitted rather than the size of the buffer for a received file.

```
  return_value = ftp_client_setup(resolve(
    REMOTE_HOST),
    REMOTE_PORT,
    REMOTE_USERNAME,
    REMOTE_PASSWORD,
    FTP_MODE_UPLOAD|PASSIVE_FLAG,
    REMOTE_FILE,
    REMOTE_DIR,
    file_buffer,
    sizeof(file_buffer));
```

```
   if (return_value != 0)  {
     printf("FTP setup failed.\n");
     exit(2);
   }
```

The `ftp_client_tick()` function returns 0 while the transfer is in progress. When the function returns 1, the transfer has completed successfully. If the function returns a value greater than 1, the transfer has failed. A message in Dynamic C's STDIO windows displays the result.

```
   printf("Looping on ftp_client_tick()...\n");
   while( 0 == (return_value = ftp_client_tick()) );

   if( 1 == return_value ) {
     printf("FTP upload completed successfully.  %d
         bytes.\n", ftp_client_filesize());
     return 0;
   } else {
     printf("FTP upload failed: status = %d, last code =
         %d\n", return_value, ftp_last_code());
     return 1;
   }
 } // end send_file()
```

TINI FTP Client

To request files from an FTP server, a TINI can use Java's `URL` and `URLConnection` classes with the *ftp* URL scheme. Applications that need to transfer files in both directions can use the TINI's `FTPClient` class.

Requesting a File in a URL

Java's `URL` and `URLConnection` classes provide support for requesting resources from remote hosts in URLs. Chapter 8 showed how a Java program can use a URL with a *mailto* protocol handler to send an e-mail. In a similar way, you can use a URL with an *ftp* protocol handler to request a file from an FTP server. The FTP capabilities are one-way only. A client can request files but can't send them.

The `FtpUrlReceiver` class below shows how a TINI can use the `URL` and `URLConnection` classes to request a file. The source code to support requesting files in URLs is in `com.dalsemi.protocol.ftp.Connec-`

tion.java in the file *ModulesSrc.jar* in the *\src* directory of the TINI distribution.

Imports and Initial Declares

The FtpUrlReceiver class imports java.io classes to support input and output functions and java.net classes to support networking functions.

```
import java.io.*;
import java.net.*;
```

A series of constant strings provide default values to use in connecting to the remote host and requesting a file. USERNAME and PASSWORD are the user name and password required to log onto the server. REMOTEHOST is the IP address or domain name of the FTP server. FILENAME is the requested file. You must change these values to match the parameters appropriate for your FTP server and requested file.

```
public class FtpUrlReceiver {

  public static final String USERNAME = "embedded";
  public static final String PASSWORD = "ethernet";
  public static final String REMOTEHOST = "192.168.111.5";
  public static final String FILENAME = "testfile.txt";
```

The FtpUrlReceiver class's constructor requires values for a remote host, user name, and password. The port variable can specify a port to use for the FTP control connection. If this value is -1, the connection uses the default port of 21. The type variable can specify a transfer type of ASCII (a) or binary (i).

```
  private String remoteHost;
  private String userName;
  private String password;
  private int port = -1;
  private String type = "a";
```

The Constructor

The class's constructor uses the passed values to set the corresponding variables.

```
  public FtpUrlReceiver(String remoteHost,
      String userName, String password) {
    this.remoteHost = remoteHost;
```

```
      this.userName = userName;
      this.password = password;
    } // end FtpUrlReceiver constructor
```

Reading a File

The class's `main()` method creates the `FtpUrlReceiver` object `ftp`. A call to the class's `getFile()` method returns the `InputStream` object `inStream`, which contains the received file. A `BufferedReader` object, `in,` reads the file from the `InputStream` object. On reading a received line of text, a `System.out.println()` statement writes the line to the default output stream. A received `null` indicates the end of the input stream. The `close()` method closes the `BufferedReader` object when the file has been read.

```
    public static void main(String[] args) {
        try {
          FtpUrlReceiver ftp = new
              FtpUrlReceiver(REMOTEHOST, USERNAME,
              PASSWORD);

          InputStream inStream = ftp.getFile(FILENAME);

          BufferedReader in = new BufferedReader(new
             InputStreamReader(inStream));
          String line;
          System.out.println("Reading " + FILENAME + ":");
          while ((line = in.readLine()) != null) {
            System.out.println(line);
          }
          System.out.println();

         in.close();

        } catch (IOException e){
          System.err.print("IO exception: ");
          System.err.println(e.getMessage());
        }
      } // end main()
```

Setting the Port and Transfer Type

The `setPort()` method can set the port to a value other than the default FTP control port of 21. Zero indicates the default port.

```
    public void setPort(int port) {
        this.port = port;
    } // send setPort()
```

The `setType()` method can set the transfer type. Use `"a"` to indicate ASCII and `"i"` to indicate binary.

```
    public void setType(String type) {
        this.type = type;
    } // end setType()
```

Requesting a File

The `getFile()` method creates and sends a URL to request a file from the FTP server. The `URL` object `url` contains the request for the file and uses the values defined earlier. If the port variable is greater than zero, the URL specifies a port. If using the default port, the URL doesn't need to specify a port value. The In Depth section of this chapter has more about the syntax of the URL.

```
    public InputStream getFile(String fileName)
        throws IOException {
      URL url = new URL("ftp://"
          + userName
          + ":" + password
          + "@" + remoteHost
          + ((port >= 0) ? (":" + port) : "")
          + "/" + fileName
          + ";type=" + type);
```

The `URLConnection` object conn reads from the FTP server referenced in the `URL` object. The `URL` object's `openConnection()` method creates the `URLConnection` object, which represents a connection to the named FTP server.

The `getInputStream()` method returns an input stream that reads from the connection to the server.

```
      URLConnection conn = url.openConnection();
      return conn.getInputStream();

    } // end getFile()
} // end FtpUrlReceiver
```

Building the Application

As with the TINI e-mail applications in Chapter 8, building the FtpUrlReceiver application requires a few additional considerations to enable using the *ftp* protocol handler. The build process uses the `com.dalsemi.protocol.*` and `com.dalsemi.protocol.ftp.*` classes in *modules.jar*.

When compiling *FtpUrlReceiver.java* to *FtpUrlReceiver.class*, you must include the location of *modules.jar* in the bootclasspath. Here is an example command line (which you can place in a batch file):

```
javac -bootclasspath ..\..\bin\tiniclasses.jar;
    ..\..\bin\modules.jar FtpUrlReceiver.java
```

When converting *FtpUrlReceiver.class* to *FtpUrlReceiver.tini,* use the BuildDependency utility in place of TiniConvertor. Here is an example command line:

```
java -classpath ..\..\bin\tini.jar;%classpath%
    BuildDependency -f FtpUrlReceiver.class
    -o FtpUrlReceiver.tini -d ..\..\bin\tini.db
    -add FTP -p ..\..\bin\modules.jar
```

The `-add` option adds the `FTP` dependency to the project, and the `-p` option names the location of `modules.jar`.

Requesting a File with FTPClient

If you need more abilities than the `URL` and `URLConnection` classes provide, the TINI's `FTPClient` class is an option. The source code for `FTPClient` is in `com.dalsemi.protocol.ftp.FTPClient.java` in the file *ModulesSrc.jar* in the *\src* directory of the TINI distribution. The class supports file transfers in both directions. The TINI's slush shell uses this class to implement an FTP client controlled via the command line.

The following example uses the `FTPClient` class to request a file from an FTP server.

Imports and Initial Declares

The `FTPClientReceiver` class imports `java.io` classes to support input and output functions. The `com.dalsemi.protocol.ftp.FTPClient` class supports communications with FTP servers.

A series of constant strings provide default values to use in connecting to the remote host and requesting a file. `USERNAME` and `PASSWORD` are the user name and password required to log on to the server. `REMOTEHOST` is the IP address or domain name of the FTP server. `FILENAME` is the requested file. You must change these values to match the parameters appropriate for your FTP server and requested file.

```
import com.dalsemi.protocol.ftp.FTPClient;
import java.io.*

public class FtpClientReceiver {
    public static final String USERNAME = "embedded";
    public static final String PASSWORD = "ethernet";
    public static final String REMOTEHOST =
        "192.168.111.5";
    public static final String FILENAME = "testfile.txt";
```

The `FtpClientReceiver` class's constructor requires values for a remote host, user name, and password. The `port` variable can specify a port to use for the FTP control connection. If this value is zero, the connection uses the default port of 21. The `type` variable can specify a transfer type of ASCII (`a`) or binary (`i`).

```
    private String remoteHost;
    private String userName;
    private String password;
    private int port = 0;
    private String type = "a";
```

The main() Method

The `main()` method creates the `FtpClientReceiver` object `ftp` using the parameters provided and calls the `doGetFile()` method to retrieve the file.

```
    public static void main(String[] args)
        throws IOException {

        FtpClientReceiver ftp = new
            FtpClientReceiver(REMOTEHOST, USERNAME,
            PASSWORD);
        ftp.doGetFile(FILENAME);
    } // end main()
```

The Constructor

The constructor uses the passed values to set the corresponding variables.

```
public FtpClientReceiver(String remoteHost,
  String userName, String password) {
    this.remoteHost = remoteHost;
    this.userName = userName;
    this.password = password;
} // end FtpClientReceiver constructor
```

Setting the Port and Transfer Type

The `setPort()` method can set the port to a value other than the default FTP control port of 21. A negative value indicates the default port.

```
public void setPort(int port) {
    this.port = port;
} // end setPort()
```

The `setType()` method can set the transfer type. Use `"a"` to indicate ASCII and `"i"` to indicate binary.

```
public void setType(String type) {
    this.type = type;
} // end setType()
```

Requesting a File

The `doGetFile()` method uses `FTPClient`'s methods to log onto the server, read responses, and request a file. For each command sent, a `System.out.println()` statement writes the returned response to the standard output stream.

The `FTPClient` object `client` specifies a non-default port if needed.

```
public void doGetFile(String filename)
    throws IOException {

    FTPClient client;
    if (port >= 0) {
      client = new FTPClient(remoteHost);
    } else {
      client = new FTPClient(remoteHost, port);
    }
```

The `userName()` and `password()` methods send the user name and password to log onto the server.

```
try {
    client.userName(userName);
    System.out.println
        (client.getResponseString());

    client.password(password);
    System.out.println
        (client.getResponseString());
```

The `ascii()` and `binary()` methods can specify whether to use ASCII or binary mode for the transfer. FTPClient also supports the methods `dir()` and `list()`. Both of these request a directory listing from the server.

```
if ("a".equalsIgnoreCase(type)) {
  client.ascii();
} else if ("i".equalsIgnoreCase(type)) {
  client.binary();
}
System.out.println
    (client.getResponseString())
```

An FTP file transfer uses two TCP connections, or channels: a control channel for commands and a data channel for the file being transferred. `FTPClient`'s `passiveConnection()` method sends a PASV command to request to use FTP's passive mode. In passive mode, the client, rather than the server, opens the data channel. FTPClient also supports the `dataConnection()` method, which uses the EPSV command to request to use extended passive mode. If the server responds that it doesn't support extended passive mode, the `dataconnection()` method sends a PORT command that specifies a port number the server should use for the data channel. The In Depth section of this chapter has more on the passive modes and PORT command.

```
client.passiveConnection();
System.out.println
    (client.getResponseString());
```

The `retr()` method sends an FTP RETR command that requests the specified file. A `BufferedReader` object reads the file, and `System.out.println()` statements write the file's contents to the standard output stream. A received null indicates the end of the input stream. The `close()` method closes the `BufferedReader` object when the file has been read.

After sending the file, the server closes the data channel. A call to `FTPClient`'s `close()` method sends an FTP QUIT command to request the server to close the control channel, which ends the session. The call to `close()` is in a `finally` block to ensure that the method is called before the `doGetFile()` method ends.

```
            client.retr(filename);
            System.out.println
                (client.getResponseString());

            BufferedReader in = new BufferedReader(
                new InputStreamReader
                (client.getDataStream()));

            String line;
            System.out.println("File contents:");
            while ((line = in.readLine()) != null) {
                System.out.println(line);
            }
            System.out.println();

            in.close();
        } finally {
            client.close();
        }
    } // end doGetFile()
} // end FtpClientReceiver
```

Building the Application

The FTPClientReceiver application uses the `com.dalsemi.protocol.ftp.FTPClient` class in *modules.jar*. So as in the previous example, when compiling *FtpClientReceiver.java* to *FtpClientReceiver.class*, you must include the location of *modules.jar* in the bootclasspath. Here is an example command line:

```
javac -bootclasspath ..\..\bin\tiniclasses.jar;
    ..\..\bin\modules.jar FtpClientReceiver.java
```

Use the BuildDependency utility to convert *FtpClientReceiver.class* to *FtpClientReceiver.tini*. Here is an example command line:

```
java -classpath ..\..\bin\tini.jar;%classpath%
    BuildDependency -f FtpClientReceiver.class
    -o FtpClientReceiver.tini -d ..\..\bin\tini.db
    -add FTP -p ..\..\bin\modules.jar
```

The `-add` option adds the `FTP` dependency to the project, and the `-p` option names the location of *modules.jar*.

Sending a File with FTPClient

In a similar way, a TINI can also use the `FTPClient` class to send a file to an FTP server. The FTPClientSender class imports `java.io.*` classes to support input and output functions. The `com.dalsemi.protocol.ftp.FTPClient` class supports communications with FTP servers.

Constant strings provide default values for the user name, password, and server's IP address. The `FILENAME` constant is the file to request from the server. You must change these values to match the parameters appropriate for your FTP server and requested file.

```
import com.dalsemi.protocol.ftp.FTPClient;
import java.io.*;

public class FtpClientSender {
    public static final String USERNAME = "embedded";
    public static final String PASSWORD = "ethernet";
    public static final String REMOTEHOST =
        "192.168.111.5";
    public static final String FILENAME = "testfile2.txt";
```

The `FtpSender` class's constructor requires values for a remote host, user name, password, and port, which is zero to specify the default port of 21. The `type` variable can specify a transfer type of ASCII (`a`) or binary (`i`).

```
    private String remoteHost;
    private String userName;
    private String password;
    private int port = 0;
    private String type = "a";
```

The main() Method

The main() method creates the `FtpClientSender` object `ftp` using the parameters provided and calls the `doSendFile()` method to send the file.

```
public static void main(String[] args)
    throws IOException {

    FtpClientSender ftp = new FtpClientSender
        (REMOTEHOST, USERNAME, PASSWORD);
    ftp.doSendFile(FILENAME);
} // end main()
```

The Constructor

The constructor for `FtpClientSender` uses the passed values to set the corresponding variables.

```
public FtpClientSender(String remoteHost,
    String userName, String password) {
    this.remoteHost = remoteHost;
    this.userName = userName;
    this.password = password;
} // end FtpClientSender constructor
```

Setting the Port and Transfer Type

The `setPort()` method can set the port to a value other than the default FTP control port of 21. Zero indicates the default port.

```
public void setPort(int port) {
    this.port = port;
} // end setPort()
```

The `setType()` method can set the transfer type. Use `a` to indicate ASCII and `i` to indicate binary.

```
public void setType(String type) {
    this.type = type;
} // end setType()
```

Sending a File

The `doSendFile()` method uses `FTPClient`'s methods to log onto the server, read responses, and send a file. For each command sent, the console displays the returned response.

The method creates the `FTPClient` object `client`, specifying a non-default port if needed.

```
public void doSendFile(String filename)
    throws IOException {

    FTPClient client;
    if (port >= 0) {
      client = new FTPClient(remoteHost);
    } else {
      client = new FTPClient(remoteHost, port);
    }

    try {
```

The `userName()` and `password()` methods send the user name and password to log onto the server.

```
        client.userName(userName);
        System.out.println
            (client.getResponseString());

        client.password(password);
        System.out.println
            (client.getResponseString());
```

The `ascii()` and `binary()` methods can specify whether to use ASCII or binary mode for the transfer. The `passiveConnection()` method requests to use passive mode for the transfer.

```
        if ("a".equalsIgnoreCase(type)) {
          client.ascii();
        } else if ("i".equalsIgnoreCase(type)) {
          client.binary();
        }
        System.out.println
            (client.getResponseString())

        client.passiveConnection();
        System.out.println
            (client.getResponseString());
```

For sending the file, the client can choose between two FTP commands. APPE requests the server to append the data being transferred to an existing file of the same name. STOR requests the server to replace any data in an existing file of the same name with the new data. With both commands, if the file doesn't exist, the server creates the file. The `issuecommand()` method can send either of these commands (or other FTP commands).

```
            client.issueCommand
                ("APPE " + FILENAME + "\r\n");
            //client.issueCommand
                ("STOR " + FILENAME + "\r\n");
            System.out.println
                (client.getResponseString());
```

An `Outputstream` object writes data to the file. A call to the class's `writeFile()` method writes the data to the `OutputStream` object. After writing the file, the output stream is flushed to send the data immediately, then closed.

After sending the file, FTPClient closes the data channel. A call to `FTPClient`'s `close()` method sends an FTP QUIT command to request the server to close the control channel, which ends the session. The call to `close()` is in a `finally` block to ensure that the method is called before the `doSendFile()` method ends.

```
            OutputStream output =
                client.getOutputStream();
            writeString(output, "test data\r\n");
            output.flush();
            output.close();
            System.out.println
                ("The file has been transferred.");

        } finally {
          client.close();
        }
    } // end doSendFile()
```

Writing a String to the Output Stream

The `writeFile()` method writes the contents of a string to an `OutputStream` object. The `stringToWrite` variable is the contents of the file to write to the server. The `String` class's `getBytes()` method converts the string to a byte array for passing to the `OutputStream` object.

```
    private void writeString(OutputStream output, String
 stringToWrite) {
        try {
          output.write(stringToWrite.getBytes());
        } catch (IOException e){
            System.err.println("IO exception: " +
e.getMessage());
         }
   } // end writeString()
} // end FtpClientSender
```

Building the Application

The FTPClientSender application uses the `com.dalsemi.protocol.ftp.FTPClient` class in *modules.jar*. So as in the previous example, when compiling *FtpClientSender.java* to *FtpClientSender.class*, include the location of *modules.jar* in the bootclasspath. Here is an example command line:

```
javac -bootclasspath ..\..\bin\tiniclasses.jar;
    ..\..\bin\modules.jar FtpClientSender.java
```

Use the BuildDependency utility to convert *FtpClientSender.class* to *FtpClientSender.tini*. Here is an example command line:

```
java -classpath ..\..\bin\tini.jar;%classpath%
    BuildDependency -f FtpClientSender.class
    -o FtpClientSender.tini -d ..\..\bin\tini.db
    -add FTP -p ..\..\bin\modules.jar
```

The `-add` option adds the `FTP` dependency to the project, and the `-p` option names the location of *modules.jar*.

Rabbit FTP Server

Dynamic C's *ftp_server.lib* library provides support for an FTP server that can enable clients to request to exchange files with a Rabbit module. The files can be stored in root memory, in the extended memory area, or in Flash memory or battery-backed RAM.

The example below shows how the Rabbit can create and serve files and enable clients to send files to the server. You can communicate with the server using any standard FTP client application or a command-line interface.

Initial Defines and Declares

As explained in Chapter 5, the firmware selects a network configuration from *tcp_config.lib*.

```
#define TCPCONFIG 1
```

A series of `define` statements configures the file system and FTP server.

The FTP server uses Dynamic C's *filesystem mk II*, also called FS2, for storing information in named files in Flash memory or battery-backed RAM. Defining `FORMAT` resets the list of files in the user block area of Flash or battery-backed memory. This statement needs to execute only the first time the program runs, to put the file system in a known state.

```
#define FORMAT
```

The `FTP_USE_FS2_HANDLERS` macro enables FS2 support in the default functions for the file handler and enables clients to write files to the file system.

```
#define FTP_USE_FS2_HANDLERS
```

The `FS_MAX_FILES` macro specifies the maximum number of files supported by the file system.

```
#define FS_MAX_FILES 50
```

The `FS2_USE_PROGRAM_FLASH` macro specifies how many kilobytes of program Flash memory the file system can use.

```
 #define FS2_USE_PROGRAM_FLASH 32
```

The `FTP_CREATE_MASK` macro provides the `servermask` parameter for the `sspec_addfsfile()` function, which makes files available to the FTP server. The default is `SERVER_FTP | SERVER_WRITABLE`, which specifies the FTP server and enables authorized users to delete and overwrite files on the server.

```
#define FTP_CREATE_MASK SERVER_FTP | SERVER_WRITABLE
```

A portion of the user block in memory holds a structure that associates the names of files with the files' locations. The `FTP_USERBLOCK_OFFSET` macro specifies the offset in the user block where the structure will be stored. The default is zero. Change this value if your application uses the default portion of the user block for another purpose. The `sizeof(server_spec)` function returns the structure's size.

```
#define FTP_USERBLOCK_OFFSET 0
```

The `SSPEC_MAXSPEC` macro specifies the maximum number of files supported by the FTP server. The default is 10.

```
#define SSPEC_MAXSPEC 10
```

The `FTP_EXTENSIONS` macro enables support for the FTP DELE (delete) command.

```
#define FTP_EXTENSIONS
```

As in the previous examples, the `#memmap xmem` directive causes all C functions not declared as root to be stored in extended memory, and the code requires the *dcrtcp.lib* library to support TCP/IP. This example also requires the *fs2.lib* library to support the FS2 file system and *ftp_server.lib* to support the FTP server's functions.

```
#memmap xmem
#use "fs2.lib"
#use "dcrtcp.lib"
#use "ftp_server.lib"
```

Starting the FTP Server

The `main()` routine performs initialization functions and starts the FTP server.

```
void main()
{
  File private_file;
  File public_file;
  FSLXnum ext;
  int file;
  int user;
  long len;
  static char create_file1_buffer[127];
  static char create_file2_buffer[127];
```

The `fs_get_flash_lx()` function returns a *logical extent number* that indicates the preferred Flash-memory device for FS2 files. The preferred device is the second Flash memory if one is available, and otherwise is the reserved area in the program Flash memory.

To use a portion of the program Flash memory for the file system, you must define two constants. In *rabbitbios.c* (in the *\bios* directory of the Dynamic C distribution), set `XMEM_RESERVE_SIZE` to the number of bytes to reserve for the file system in the program Flash. And in your application, before the statement `#use "fs2.lib"`, define `FS2_USE_PROGRAM_FLASH` to equal the number of kilobytes the file system will use. The application uses the smaller of the two values. (This application sets `FS2_USE_PROGRAM_FLASH` to 32 kilobytes above.)

To use battery-backed RAM for the file system, use the `fs_get_ram_lx()` function in place of `fs_get_flash_lx()`. To use the non-preferred Flash memory for the file system, use the `fs_get_other_lx()` function.

```
   ext = fs_get_flash_lx();
```

The `fs_init()` function initializes the file system.

```
   fs_init(0, 0);
```

If the `FORMAT` macro is defined in the application, the file system and user block are initialized to known states. The `lx_format()` function formats the file system extent, deleting any files that were present. The `writeUser-Block()` function initializes the user block in memory to zeros. Don't execute this block of code if you want to preserve files already in memory.

```
 #ifdef FORMAT
   lx_format(ext, 0);
   len = 0;
   writeUserBlock(FTP_USERBLOCK_OFFSET, &len,
       sizeof(long));
 #endif
```

The application creates two FS2 files and stores text in each. File 1 is `public_file` and contains the text "public file". File 2 is `private_file` and contains the text "private file".

```
    sprintf(create_file1_buffer, "public file");
    fcreate(&public_file, 1);
   fwrite(&public_file, create_file1_buffer,
       strlen(create_file1_buffer));
   fclose(&public_file);

   sprintf(create_file2_buffer, "private file");
   fcreate(&private_file, 2);
```

```
fwrite(&private_file, create_file2_buffer,
    strlen(create_file2_buffer));
fclose(&private_file);
```

The `ftp_load_filenames()` function loads the data structure that keeps track of the locations of the files. On success, `ftp_load_filenames()` returns zero and a call to `ftp_save_filenames()` saves the data structure to the user block. Even if there are no file names defined yet, saving the data structure puts it in a known state.

```
if (ftp_load_filenames() < 0) {
  ftp_save_filenames();
}
```

The application uses Dynamic C's `ServerSpec` structure defined in *zserver.lib* and introduced in Chapter 7. The `sauth_adduser()` function defines a user who can access files on the server. A file can be accessible to a specific user or users, or to any user.

To make a file accessible to all users, define a user with the user name of `anonymous` and an empty string (`""`) for a password. The `SERVER_FTP` parameter names the server that the user can access. The `ftp_set_anonymous()` function specifies the user name for files that anyone can access.

The `sspec_addfsfile()` function enables the FTP server to access the FS2 files created earlier. The function associates the file name *public.txt* with file 1 on the FTP server. The `sspec_setuser()` function enables the `anonymous` user defined above to access the file.

```
user = sauth_adduser("anonymous", "", SERVER_FTP);
ftp_set_anonymous(user);

file = sspec_addfsfile( "public.txt", 1, SERVER_FTP ) ;
sspec_setuser(file, user );
```

In a similar way, the following statements define a user with the user name `rabbit1` and password `embedded`. The `sauth_setwriteaccess()` function enables rabbit1 to send files to the server in addition to requesting files. Rabbit1 can access file 2 on the server as the file *private.txt*. This file isn't available to anonymous users.

```
user = sauth_adduser("rabbit1", "embedded",
       SERVER_FTP);
   sauth_setwriteaccess(user, 1);

   file = sspec_addfsfile( "private.txt", 2, SERVER_FTP );
   sspec_setuser(file, user );
```

A call to `sock_init()` initializes the TCP/IP stack. A call to `ftp_init(NULL)` initializes the FTP server using the default handlers. As explained in Chapter 6, calling `tcp_reserveport()` can improve the server's performance. The `FTP_CMDPORT` constant is 21, the default FTP command port.

An endless loop calls `ftp_tick()` to process FTP requests as needed.

```
   sock_init();
   ftp_init(NULL);

   tcp_reserveport(FTP_CMDPORT);

   while(1) {
     ftp_tick();
   }
 } // end main()
```

TINI FTP Server

The TINI software includes an FTP server. When the *.startup* file in the TINI's */etc/* directory contains this line:

```
setenv FTPServer enable
```

the slush shell runs the FTP server on start up. This is the server you use to transfer *.tini* programs to the TINI. You can use the same server to transfer files in both directions for any purpose using a standard FTP client application.

The TINI's `FTPServer` class assumes that slush is present and uses some of its commands. For example, on receiving an FTP LIST command from a client, the server tries to invoke slush's `ls` command. Other slush commands that the FTP server might call include `cd`, `cd ..`, `del`, `ls`, `ls -l`, `md`, `move`, and `rd`. If you want to use the TINI's FTP server in your application, the TINI will need to also be running slush or another shell that implements the above commands.

The source code for the FTP server is in the `com.dalsemi.shell.server.ftp` classes `FTPServer.java`, `FTPSession.java`, and `FTPInputStream.java`. These are in the file *APIsrc.jar* in the *\src* directory of the TINI distribution.

In Depth: Inside the File Transfer Protocol

The File Transfer Protocol defines a standard protocol for transferring files between computers. The main documents that define FTP are *RFC 959: File Transfer Protocol (FTP) and RFC 1123: Requirements for Internet Hosts -- Application and Suppo*rt.

Requirements

An embedded system can function as an FTP client or server. A client initiates communications with a server and sends requests to transmit or receive files. In most cases, an embedded system that needs to exchange files with a single PC should function as a client. Many embedded systems don't have a lot of resources to spare, and running an FTP server that is always available requires processing time and memory. Running a server also puts the system at a greater security risk because any computer in the network might be able to gain access to the system's files. But if the embedded system needs to make its files available to anyone on the network, or if the files need to be available to other computers at all times, the system will need to function as a server.

A computer that uses FTP must have a file system, which enables the system to store information in named entities called files. Files are of course useful in desktop computers, where you select files to run programs, view documents and images, and perform other tasks.

Embedded systems can support file systems as well. A small embedded system may just store data in specified locations in memory, with no need to place the data in named files. But for many embedded systems, a file system provides a useful structure for accessing information, both locally and over a network.

For example, a system can store collected data or configuration settings in files. A system functioning as an FTP client can initiate communications periodically with a remote computer to request to send or receive files. A system functioning as an FTP server can make its files available on request and can allow remote computers to send files that the system will use. The user that communicates with the embedded system can be a human using an FTP program or a process that functions without human intervention. For example, a PC can be programmed to retrieve a file once a day from an embedded system.

In PCs, the file system includes the ability to store files in a directory structure and to specify attributes such as whether a file is write-protected or accessible to certain users. Under Windows XP, from the **My Computer** folder, you can browse the directories and view file names and attributes. (In the **View** menu, click **Choose Details** to specify what information to display and click **Details** to view the information.) The TINI supports a similar file system, which you can browse from the slush shell using commands such as `ls -l` and `cd`.

A very basic file system might just consist of a structure with a series of entries that each store the name, starting address in memory, and length of a file. In Dynamic C, entries in an `HttpSpec` or `ServerSpec` structure can specify files that are accessible to a Web or FTP server. Each entry includes a file name, the address in memory where the file's length and contents are stored, and optional security information.

On a PC, you can perform FTP transfers using an FTP client application such as WS_FTP from Ipswitch, Inc. Two other ways to perform FTP transfers are from a command prompt and from a browser. To use the command-line interface, enter `ftp` at a command prompt and enter `?` for a list of commands. The browser interface is explained later in this chapter. Ipswitch and others also offer applications that enable a PC to function as an FTP server.

Transferring a File

To transfer a file, an FTP session uses two channels, or communications paths, one for control information and one for the file being transferred. Each channel has a separate TCP connection.

On the server, the default port for the control channel is 21 and the default port for the data channel is 20. The client can use any available port or ports. The default for the client is to use the same port for both the control and data channels. However, transfers that use FTP's stream mode, which requires a new data connection for each file, should send a PORT command to specify a new, non-default port for each file transfer.

Requesting a new port for each transfer prevents problems due to TCP's timeout requirements. When a connection closes, TCP requires a timeout before the same connection can be reused. The timeout prevents a new connection that is identical to a recently closed connection from receiving data intended for the previous connection. When transferring multiple files in a single session, if a transfer tries to use the same port as the previous connection, the port may be unavailable because thes timeout for the previous connection hasn't expired. Specifying a different port for each data connection eliminates the problem. Other alternatives are to use the block or compressed transfer modes, which don't require a new data connection for each file.

These are typical steps in sending a file to a server in stream mode, where the file's contents are sent without a header or any assumed structure for the file's data:

1. The client opens a control channel between any available local port and port 21 on the server. The client sends commands to establish communications and request to send a file.

2. The server opens a data channel between the server's port 20 and the port the client is using for the control channel.

3. The client sends the file's contents, closes the data channel, and requests the server to close the control channel.

4. The server closes the control channel.

In a similar way, these are the steps in receiving a file from a server in stream mode:

1. The client opens a control channel between any available local port and port 21 on the server. The client sends commands to establish communications and request a file.

2. The server opens a data channel between the server's port 20 and the port the client is using for the control channel. The server sends the file and closes the data channel.

3. The client requests the server to close the control channel.

4. The server closes the control channel.

A client that is communicating from behind a firewall may find that the firewall blocks the server's request to open the data connection. To get around this limitation without having to reconfigure the firewall, the client can send a command that requests a passive transfer process (PASV or EPSV), where the client, rather than the server, opens the data connection. The client must send the command to request a passive transfer preceding each transfer.

When a client specifies the location of a file on a server, the location is relative to the directories that the server makes available to the client. This location can differ from the file's absolute location in the computer. For example, a computer functioning as a server may allow the user to access the directory */ftp/user1* and its subdirectories. The server's root directory for that user is then */user1*. To access a file at */ftp/user1/data/test.txt*, the client would specify the location on the server as */data/test.txt,* which is the file's location relative to the user's root directory.

Commands

The FTP standard defines required and optional commands for FTP servers to support.

All of the commands and symbols that represent parameter values are case-insensitive. A command ends with CRLF.

Minimum Implementation

RFC 959 specifies the commands that a minimum implementation of FTP must support, and RFC 1123 updates this list with additional commands. The implementation specified by RFC 1123 is more capable in handling communications between computers that may use different operating systems, file systems, and firewall protection.

However, RFC 1123 says that computers whose operating system or file system doesn't allow or support a command aren't obligated to add support for it. So for example, an embedded system whose file system doesn't support subdirectories can run an FTP server that doesn't support MKD, CWD, or other commands that manipulate directories.

In reality, which commands a system's software needs to support depends in part on how the system will use FTP. On a PC, a user that needs to exchange files with varied FTP servers will want an FTP client application that is as capable and flexible as possible. And an FTP server that is available to varied clients will want to support a large command set. But an embedded system that exchanges files only with known FTP clients or servers can have a more minimal implementation. If the transfers are only with known servers or clients and are controlled entirely by software at both ends, the commands can be known, predictable, and thus limited.

The following commands are the minimum implementation required by RFC 1123, plus EPSV and EPRT, which have additional support for IP v6 addresses. The commands included in RFC 959's smaller subset are noted as well.

ACCT *account*

The ACCT command identifies a user account. A server may require an ACCT value to log on, or a system may use accounts to grant specific privileges (to store files, for example) at any time after logging on.

APPE *pathname*

With the APPE command, the client requests the server to append the received data to the named file if it exists, and otherwise to create the file and store the received data in it.

CDUP

The CDUP command requests to change to the current directory's parent directory.

CWD *pathname*

The CWD command requests to change the working directory to the directory specified in *pathname*.

DELE *pathname*

The DELE command requests to delete the file specified in *pathname* on the server.

EPSV

The EPSV command requests the server to wait for the client to open the data connection instead of having the server open the connection. The server responds to this request with code `227 entering extended passive mode`, followed by the port number where the server will listen for the client. The format of the response is:

Entering Extended Passive Mode (|||*port_number***|)**

where *port_number* is the number of the port the server will be listening on. The recommended delimiter character is ASCII 124 (|). The first two fields are place holders for future use and must be empty. The format is similar to the format of the argument passed with EPRT, described below.

This command is defined in *RFC 2428: FTP Extensions for IPv6 and NATs*. Also see the PASV command. Many servers support PASV, but not EPSV.

EPRT

The EPRT command enables the client to provide an extended address for the data connection.

The format of EPRT is:

EPRT |*net-prt***|***net-addr***|***tcp-port***|**

where:

net-prt is an Address Family Number from the list maintained by IANA. IP Version 4 is 1; IP Version 6 is 2.

net-addr is the IP address. IP Version 4 addresses use dotted quad notation. IP Version 6 addresses use the representation described in *RFC 2373: IP Version 6 Addressing Architecture*.

tcp-port is the number of the TCP port where the host is listening for a connection.

This command is defined in *RFC 2428: FTP Extensions for IPv6 and NATs*. Also see the PORT command. Many servers support PORT, but not EPRT.

HELP [*command name*]

The HELP command requests text that explains how to use the server or how to use an optional command provided as a parameter with the command.

LIST [*pathname*]

The LIST command requests the server to send a list of files in the directory specified in *pathname* or information about the file specified in *pathname*. If there is no parameter sent with the command, the server returns information about the current directory.

MKD *pathname*

The MKD command requests to create a directory specified in *pathname* on the server.

MODE *mode*

The MODE command specifies a transfer mode: stream (s), block (b), or compressed (c). In stream mode, the default, the data has no assumed format. In the optional block and compressed modes, the data begins with a header that enables the receiver to determine when a transfer is complete, so there's no need to close the data connection after each transfer to indicate end of file. Compressed mode also enables sending compressed data for faster transfers.

RFC 959's minimum implementation requires support for stream mode.

NLST [*pathname*]

The NLST command requests the server to send a list of file names in the directory specified in *pathname*.

NOOP

The NOOP command performs no function except to elicit a response that confirms that the server is responding to commands.

RFC 959's minimum implementation includes support for NOOP.

PASS *password*

With the PASS command, the client specifies the password for the user name. If the user name is `anonymous`, the password conventionally is the user's e-mail address.

PASV

The PASV command requests the server to wait for the client to open the data connection instead of having the server open the connection. The server responds to this request with the code `227 entering passive mode`, followed by the IP address and port number where the server will listen for the client. This information uses the same format as the PORT command. Passive mode can be useful when communicating through firewalls. Also see EPSV.

PORT *host-port*

The PORT command enables the client to specify an IP address and port number the client will use for the data connection. The *host-port* parameter consists of four decimal numbers that represent the four bytes that make up a 32-bit IP address, followed by the two bytes of the port address. The parameter uses the format *h1,h2,h3,h4,p1,p2*, where *h1* is the high byte in the IP address followed by the next three bytes in order and *p1* is the high byte in the port number, followed by the low byte.

For example, to request to use port 53249 (D001h) at IP address 192.168.111.100, the command would be `PORT 192,168,111,100,208,1`. (The decimal value 53249 equals D001h. D0h is 208 in decimal, so the decimal values of the two bytes are 208 and 1.)

As explained above, issuing a PORT command before establishing a data connection can prevent problems due to TCP's timeout requirements. Transfers that use passive or extended-passive mode don't require a PORT command because the server waits for the client to connect on the port the server has specified.

RFC 959's minimum implementation includes support for PASV.

PWD

The PWD command prints the name of the current working directory.

QUIT

With the QUIT command, the client requests the server to close the control connection. If the data connection is open, the server will wait for it to close before closing the control connection.

RFC 959's minimum implementation includes support for QUIT.

RETR *pathname*

A client uses the RETR command to request a file from the server. The *pathname* parameter specifies the file's path, if needed, and name.

RFC 959's minimum implementation includes support for RETR.

RMD *pathname*

The RMD command requests to remove a directory specified in *pathname* on the server.

STAT [*path*]

The STAT command requests status information. If the command has no parameter, the server returns the current values of all transfer parameters and the status of connections. If the command includes a path, the command returns a directory listing for the path, as in a LIST command, but using the control connection.

STOR *pathname*

A client uses the STOR command to request to send a file to the server. The *pathname* parameter specifies the file's path, if needed, and name. If the file doesn't already exist on the server, the server creates the file. If the file does exist on the server, the server overwrites the file.

RFC 959's minimum implementation includes support for STOR.

STRU

The STRU command specifies the structure of the data's contents. The file structure (f), which is the default, makes no assumptions about the structure of the data. With the record structure (r), the data is assumed to consist of sequential records in a prescribed format.

RFC 959's minimum implementation includes support for the file structure and support for the record structure if the file system supports records.

SYST

The SYST command returns text that indicates what operating system the server is running. Standard text to use for popular operating systems is available at *www.iana.org/assignments/operating-system-names*. The options include WIN32 and more specific designations such as WINDOWS-98 and WINDOWS-CE.

TYPE

The TYPE command can specify how text characters are encoded in the files being transferred. In ASCII Non-print (AN), which is the default, a character is represented by a byte containing a 7-bit NVT-ASCII code, with the most significant bit set to zero. The Telnet standard (*RFC 854: Telnet Protocol Specification*) defines the NVT-ASCII character set, which includes codes for carriage return (CR) and line feed (LF). Non-print means that the data isn't required to include vertical-format information such as CRLF or page breaks. Other FTP types are EBCDIC and Image.

RFC 959's minimum implementation includes support for ASCII Non-print type.

USER *username*

In the USER command, *username* identifies the client requesting access to the server's resources. When a server is available to any client, *username* is `anonymous`.

RFC 959's minimum implementation includes support for USER.

Additional Commands

RFC 959 defines additional commands and valid reply codes, and RFC 2228: *FTP Security Extensions* adds more. On receiving an unrecognized command, a server returns reply code 502 (Command not implemented).

Requesting a File with a URL

A computer that only needs to receive files, but not send them, can use a URL to communicate with an FTP server. The URL standard (RFC 1738) defines an *ftp* scheme for URLs. The scheme is:

ftp://*user***:***password***@***host*[**:***port*]**/***url-path*

where

user is the user name to gain access to the FTP server.

password is the user's password. If the URL doesn't supply a password, a browser may prompt for it.

host is the host's IP address in dotted-quad format or a domain name.

port is the port to connect to on the server. The port is 21 if not specified.

url-path is the location and name of the file being requested.

The url-path is in the form:

[*cwd1***/***cwd2***/***...cwdn*]**/***filename*[**;***type*=typecode]

where *cwd1*, *cwd2*, and so on are any directories required to specify the location of the file on the server, *filename* is the name of the file being requested, and an optional *typecode* specifies the type of resource being requested. A `typecode` of `a` means ASCII Non-print, which is the default if not specified. A `typecode` of `i` is Image (binary), and `d` means directory. Requesting a URL for an ASCII or image file causes the client to send a RETR command for the named file. Requesting a URL that names a directory causes the client to send an NLST command to request a list of files in the specified directory.

The browser or other software that supports the FTP scheme opens a connection with the specified host, and sends the appropriate FTP commands to retrieve the file or list of file names.

In Java, an instance of the `URL` class represents a URL. As the TINI example in this chapter showed, an instance of the `URLConnection` class can communicate with a resource that a URL references, such as an FTP server.

10

Keeping Your Devices and Network Secure

If your device connects to the Internet, you need to pay attention to network security. Many devices that connect only to local networks can benefit from security measures as well.

Without effective security, an unauthorized user may do any of the following:

- View your data, device firmware, or the contents of any files.
- Alter or erase files.
- Install and run program code on your device.
- Submit Web-page form data that causes the device to malfunction or has other unintended consequences.
- Spy on transmissions to and from your device.
- Gain access to other computers in the local network.

10

Keeping Your Devices and Network Secure

If your device connects to the Internet, you need to pay attention to network security. Many devices that connect only to local networks can benefit from security measures as well.

Without effective security, an unauthorized user may do any of the following:

- View your data, device firmware, or the contents of any files.
- Alter or erase files.
- Install and run program code on your device.
- Submit Web-page form data that causes the device to malfunction or has other unintended consequences.
- Spy on transmissions to and from your device.
- Gain access to other computers in the local network.

- Clog your network with repeated attempts to communicate, preventing authorized users from accessing the device and other computers in the local network and possibly preventing the device from performing the tasks it's responsible for.

Fortunately, there are steps you can take to prevent these activities. Not every device needs to implement every security measure. What steps to take depend on the device, its capabilities and responsibilities, the local network the device resides in, and any connections the device has to networks outside the local network.

In some ways, embedded systems are often inherently more secure than a PC with a familiar operating system and plenty of resources to exploit. If your device's firmware is in a one-time-programmable (OTP) ROM, you don't have to worry about preventing malicious users from overwriting the firmware. If your device serves Web pages that contain no private information, there's no need to encrypt the data being sent. But in most cases, there are risks you need to protect against, to ensure that your device continues to operate as it should and to ensure that the security of other computers in the local network aren't compromised.

One way to limit who has access to a resource is to require a user name and password before serving the resource. This chapter shows how you can use HTTP's Basic Authentication to protect resources with user names and passwords. The In Depth section details four steps that will go a long way to ensuring the security of your devices and the local networks they reside in.

Quick Start: Limiting Access with Passwords

For many applications, it's desirable to limit access to certain Web pages by requiring users to enter a valid user name and password. HTTP 1.0 supports Basic Authentication, which enables a server to require a valid user name and password before returning a Web page.

Basic Authentication is sufficient protection for some applications, and many networking libraries and packages for embedded systems support it.

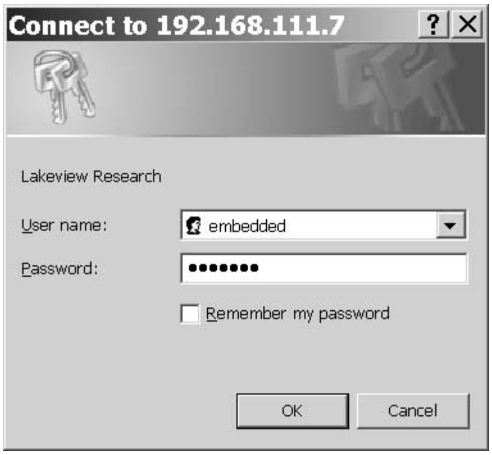


Figure 10-1: On receiving a request for Basic Authentication, browsers display a window like this to enable users to enter a user name and password.

Using Basic Authentication

When a client requests a Web page protected with Basic Authentication, the server requests the client to authenticate, or prove that the client is authorized to receive the resource. The server does this by returning an HTTP header with the error code 401 (Unauthorized) and a WWW-Authenticate field that names the type of authentication required. Here is an example:

```
HTTP/1.0 401 Unauthorized\r\n
Date: Mon, 14 Apr 2003 12:05:15 GMT\r\n
WWW-Authenticate: Basic realm="Embedded Ethernet" \r\n
\r\n
```

The WWW-Authenticate field names two values: the authentication scheme, or method, to use (`Basic` in the example) and the realm the scheme applies to (`"Embedded Ethernet"`). On returning a valid user name and password in the format required by the named authentication scheme, a client can access resources within the named realm. A server can support multiple realms, with each allowing access to a different set of users.

On receiving a header requesting Basic Authentication, the client's browser typically displays a window that requests the user to enter a user name and password. Figure 10-1 shows an example. The window displays the name of the page's realm, so when naming a realm, use something meaningful to end users. For added security, most browsers display dots in place of the password's characters when they're entered. When the user has entered the requested information and clicks OK, the browser sends an authorization request containing the password and user name. The request travels in an HTTP GET request that includes an Authorization request with the encrypted user name and password:

```
GET / HTTP/1.0\r\n
Authorization Basic ZW1liZWRkZWQ6ZXRoZXJuZXQ/r/n
/r/n
```

On receiving the GET request, the server decrypts the user name and password. If both are valid for the specified realm, the server returns the Web page originally requested. If not, the server typically returns another response with error 401 and a request to authenticate. Most browsers display the authentication window again, but after receiving a third request to authenticate, some browsers give up and don't re-display the window. Opening a new browser window typically allows the user to try again, however.

The encryption used in Basic Authentication is the Base64 Content-Transfer-Encoding method described in *RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, minus the specified limit of 72 characters per line.

In the encoding, the data to transmit is first divided into 24-bit chunks. Each chunk is then divided into four 6-bit numbers. A table provided in the standard assigns a character in the BASE64 alphabet to each 6-bit value (0 to 63). The BASE64 alphabet includes upper- and lower-case letters, numerals, and a few additional characters. For example, binary 000000 in BASE64 is the character *A*, and binary 011010 (26 decimal) is the character *a*.

In a request for Basic Authorization, the client converts a string in this format:

*user_name***:***password*

to BASE64. The resulting characters transmit in the Authorization field of the HTTP header.

For example, if the user name is `embedded` and the password is `ethernet`, the string to encrypt is:

```
embedded:ethernet
```

Each character is a byte, so there are 17 bytes, which equal 22 6-bit values with four bits left over. To obtain an integral number of 6-bit values, pad the end with two zeroes. Encrypting the user name and password gives this 23-character string:

```
ZW1iZWRkZWQ6ZXRoZXJuZXQ
```

The result must be an integral multiple of 24 bits. When needed, add one or two equal signs (=) to the end of the string to lengthen it. The example above requires one equal sign.

The BASE64 encryption can be easily decrypted by anyone spying on a transmission. It's also possible for a determined hacker to keep trying different user names and passwords until something works. The In Depth section of this chapter describes Digest Authentication, which is more complex but more secure and thus more suitable for some applications.

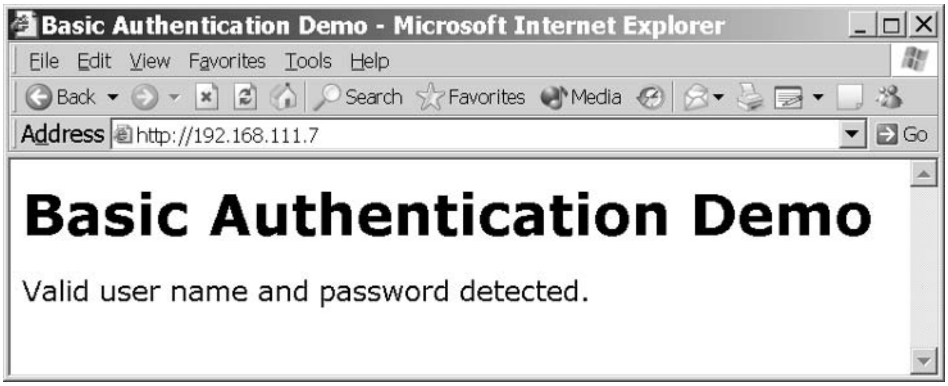
Basic Authentication on the Rabbit

Rabbit Semiconductor's *http.lib* library includes support for Basic Authentication. Chapter 6's example introduced the `HttpSpec` structure, which contains an `HTTP_FILE` entry for each file a Rabbit's Web server can access. Each entry can also specify a realm for password-protecting the file.

To protect a file, the application must include an `HttpRealm` structure with one or more user names and passwords, and the file's `HTTP_FILE` entry must specify the realm, as in the following example application.

Initial Defines and Declares

Much of the code that configures and initializes the Rabbit is the same as in previous examples in Chapter 6 and Chapter 7, so I'll skip extended explanations of these statements.


Basic Authentication Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://192.168.111.7
Go

Basic Authentication Demo

Valid user name and password detected.

Figure 10-2: On receiving a valid user name and password, the application returns the requested Web page.

```
#define TCPCONFIG 1
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
```

An `#ximport` directive imports a Web page (*index.html*) that displays a message on successful authentication.

```
#ximport "c:/rabbit/passworddemo/index.html"
    index_html
```

Figure 10-2 shows an example Web page. In a real-world application, this page would display the protected contents.

The `HttpRealm` structure `myrealm` contains a single entry that defines an authorized user with a user name (`"embedded"`), password (`"ethernet"`), and realm name (`"Lakeview Research"`):

```
const HttpRealm myrealm[] =
{
   {"embedded", "ethernet", "Lakeview Research"}
};
```

The single entry in the `HttpType` structure associates the file extension *.html* with the handler for files of type `text/html`.

```
const HttpType http_types[] =
{
   { ".html", "text/html", NULL}
};
```

The `HttpSpec` structure contains information about the file the server serves. The two entries enable clients to request the file by name (`"/index.html"`) or as the default file served on entering the server's IP address alone (`"/"`) in a browser's Address text box.

```
const HttpSpec http_flashspec[] =
{
   { HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL,
     myrealm},
   { HTTPSPEC_FILE, "/index.html", index_html, NULL,
     0, NULL, myrealm}
};
```

The main() Function

The `main()` function initializes the TCP/IP stack and the HTTP handler and calls `tcp_reserveport()` to enable establishing a connection even if no sockets are available. An endless `while` loop then calls `http_handler()` repeatedly to handle any incoming requests.

```
main()
{

   sock_init();
   http_init();
   tcp_reserveport(80);

   while (1) {
       http_handler();
     }
} // end main()
```

When the program is running and a client requests *index.html* or the server's default file, the server returns an authentication request. On receiving an authentication request from the client with the required encrypted user name, password, and realm, the server returns the Web page *index.html*.

In a similar way, you can use Basic Authentication in Rabbit applications that use forms and the *zserver.lib* library, as described in Chapter 7. Rabbit Semiconductor has an example application that illustrates Basic Authentication with forms.

Basic Authentication on the TINI

For the TINI, Web servers that support Java servlets, such as the Tynamo Web server and TiniHttpServer, typically support Basic Authentication as well. The following `BasicAuthentication` servlet for the Tynamo Web server requires clients to provide a valid user name and password before the servlet will serve its Web page to the client.

Initial Imports

As in the previous example, the name of the realm is "Lakeview Research," the user name is "embedded," and the password is "ethernet."

In addition to the `java.io.IOException`, `javax.servlet,` and `javax.servlet.http` classes, the servlet imports the `AuthenticatedHt-tpServlet` class from Tynamo's `com.qindesign.servlet` package.

```
import java.io.IOException;
import javax.servlet.ServletOutputStream;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.qindesign.servlet.AuthenticatedHttpServlet;

public class BasicAuthentication extends
    AuthenticatedHttpServlet {
```

Methods

The `getRealm()` method returns the name of the realm. The class must support this method.

```
    public String getRealm(HttpServletRequest req) {
        return "Lakeview Research";
    }
```

The `isAuthorized()` method is passed a realm, user name, and password and checks to see if these match the values supported by the servlet. The class must support this method.

```
  public boolean isAuthorized(String realm,
      String username, String password) {
    return "Lakeview Research".equals(realm) &&
        "embedded".equals(username) &&
        "ethernet".equals(password);
    } // end getRealm()
```

The `doGet()` method functions like the `doGet` methods in previous examples, except that it is called only after a GET request has been authorized. In this example, the method returns a Web page containing a single line of text informing the client that the user name and password are valid. In a real-world application, this page would display the protected information.

```
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException {
  resp.setContentType("text/html");
  ServletOutputStream out = resp.getOutputStream();
      out.println("<P>Valid username and password
      detected.");
} // end doGet()
```

In a similar way, the servlet can respond to authorized POST requests. This example just calls the `doGet()` method.

```
protected void doPost(HttpServletRequest req,
  HttpServletResponse resp)
        throws ServletException, IOException {
    doGet(req, resp);
} // end doPost()
```

The `doUnauthorizedGet()` method is like `doGet()`, except that it's called when an authorization attempt fails. The method writes the contents of a Web page to a `ServletOutputStream` object. The Web page contains an error message and a link that requests the servlet again to give the user another chance.

Users may not see this Web page every time they send a request that fails authentication. On receiving an HTTP response with error code 401 (Unauthorized), many browsers display the authentication window again and ignore the Web page returned in the response. But if the browser gives up after three tries, or if the user closes the authentication window without clicking OK, the browser may display the Web page.

```
protected void doUnauthorizedGet(HttpServletRequest
    req, HttpServletResponse resp)
    throws ServletException, IOException {
  resp.setContentType("text/html");
```

```
            ServletOutputStream out = resp.getOutputStream();
            out.print("<P>Invalid username or password.");
            out.print("<P><A HREF=
                \"/servlet/BasicAuthentication\">Try
                again</A>");
          } // end doUnauthorizedGet()
```

In a similar way, the servlet can respond to unauthorized POST requests. This example just calls the `doUnauthorizedPost()` method.

```
      protected void doUnauthorizedPost (HttpServletRequest
          req, HttpServletResponse resp)
                throws ServletException, IOException {
        doUnauthorizedGet(req, resp);
      } // end doUnauthorizedPost

    } end BasicAuthentication
```

When this servlet has been compiled and the Tynamo Web server is running, requesting the servlet */servlet/BasicAuthentication* at the TINI's IP address will cause Figure 10-1's window to display. On receiving a GET request with the user name "embedded" and password "ethernet" using Basic Authentication, the `doGet()` method returns the Web page with protected content to the client.

In Depth: Four Rules for Securing Your Devices and Local Network

Paying attention to the following four rules will go a long way in ensuring that your device, data, and local network are as secure as possible from security risks:

1. Use a **firewall** and configure it with the most restrictive settings that allow your device to perform the communications it requires.

2. Restrict access to individual protected resources with **user names and passwords**.

3. **Validate** data provided by users to ensure the contents won't cause harm.

4. **Encrypt** data that must remain private.

For each of these, you need to review the risks as they apply to your device, then take actions as needed to reduce or eliminate the risks. The actions will vary with the device, the firmware, and the security needs of the computers in any local network the device attaches to.

Use a Firewall

A firewall is the first line of defense against unauthorized access to the resources of your device and local network. Chapter 4 introduced firewalls and explained the need to configure them to allow a device to function as a server on the Internet. This chapter has more about firewalls, including how to select and use a firewall to provide the maximum protection for your device and local network while still allowing necessary communications to pass through the firewall.

Three ways for an embedded system to obtain firewall protection are a dedicated firewall device, firewall software running on a PC in the same local network as the embedded system, and firewall firmware in the device itself. A dedicated device is the easiest to use. Firewall software in a PC has the advantage of costing nothing if you have a PC available and running that can function as a firewall for a local network. Firmware that performs the function of a firewall in the device can be an option in some cases where you need to protect a single device.

Firewall Basics

A firewall device is an embedded system that connects between a local device or network and the Internet or other networks the local computer(s) communicate with. The firewall typically has multiple LAN ports for connecting local computers and hubs and a single WAN port that connects to the outside world. The local computers are said to be *behind the firewall.* Everything the WAN port can communicate with is *outside the firewall.* In smaller networks, the WAN port often connects to a cable or DSL modem that connects to an ISP. Every communication to or from a computer outside the firewall must go through the firewall to reach a computer in the local network. The firewall's configuration determines which communications can pass through the firewall.

Firewalls are mainly concerned with restricting incoming communications, though in some cases, a firewall may also block outgoing communications that appear to be fraudulent, such as an outgoing datagram with a non-local Source Address.

Many firewall devices are multi-function devices that also perform the functions of a hub and a router with network address translation (NAT). (See Chapter 4 for more about NAT.) The hub enables multiple computers to connect to the firewall. To add more computers, you can connect another hub to one of the local ports as described in Chapter 2.

In a similar way, a Windows XP PC configured to use Internet Connection Sharing (ICS) can protect a local network, including embedded systems, by enabling Windows XP's Internet Connection Firewall. The PC must have two network interfaces. An Ethernet interface connects the PC to the local computers protected by the firewall. A second Ethernet interface or an interface to a modem connects the PC to the world outside the firewall. The Internet Connection Firewall has configuration options similar to those for a dedicated firewall device.

A firewall's configuration determines which IP datagrams the firewall will allow to pass through to the local network. Most firewall devices support a password-protected Web interface for setting the configuration. To configure the firewall device, you need a network-connected PC or other computer that enables you to view and enter information on the Web pages, but once the firewall is configured, the device protects the network without requiring a connected PC. For added security, many firewalls enable you to restrict access to the configuration pages to computers in the local network only.

The specifics of how to configure a firewall vary with the manufacturer and model, but the general concepts are the same for all firewalls. The basic rule for configuring a firewall is to block all communications through the firewall except those that you explicitly want to allow.

Functioning as a Client

Some embedded systems can function strictly as clients that request resources from or send data to other computers but don't have to accept communications from hosts the client hasn't initiated communications with. For example, a system that uses the Internet only to send periodic sensor readings to remote computers doesn't need to accept communications from computers other than the ones the system sends the reading to. The firewall can examine each datagram received from outside the firewall. If the information in the headers shows that the datagram's source and destination match those of a valid, currently active connection, the datagram can pass through to the local network. If not, the firewall drops the datagram and may return a response indicating that the data was refused.

To help in deciding whether to allow a received datagram to pass to the local network, the firewall may maintain and consult a table that contains an entry for each connection. When a local computer sends a TCP segment or UDP datagram to a remote host and port, a firewall can create a table entry that allows incoming traffic from that remote host and port to pass to the specified local host and port. For TCP connections, the firewall deletes the entry when the TCP connection is closed as indicated by the FIN or RST flag. For UDP, which doesn't use formal connections, the firewall can use a timeout to decide when to delete the entry. TCP connections can also use a timeout as a backup for cases where the connection doesn't close properly.

As Chapter 9 explained, in FTP transfers, by default the server requests to open a TCP connection for a transfer's data channel. If the client's firewall blocks requests to open a connection, the client can request to use passive or extended passive mode, where the client computer opens the connection using a port number provided by the server.

Hosting a Server

If a local computer needs to be able to serve resources to requesting computers outside the firewall, you need to configure the firewall to allow the requests to pass through the firewall while preventing other, unwanted traffic from entering the local network.

A firewall may allow several options for restricting incoming traffic. For example, a local network might include an embedded system that hosts a Web server on port 80, the default port for HTTP communications. Configuration options for allowing incoming HTTP requests include the following, from most restrictive to least restrictive:

- Allow incoming IP datagrams that don't belong to an established connection only if they contain TCP segments that contain HTTP requests that are directed to port 80, and forward the TCP segments to a specified host. This is the most secure option. A datagram passes through the firewall only if the datagram contains a TCP segment, the contents of the segment's Destination Port Number field is 80, and the contents of the segment's data area indicate that the message is an HTTP request. Not all firewall devices are capable of filtering in this much detail. Also, additional fragments in a fragmented datagram won't have a TCP or HTTP header to examine, so the firewall needs to have a mechanism that allows additional fragments to pass through the firewall.
- Allow incoming IP datagrams that don't belong to an established connection only if they contain TCP segments directed to port 80. Forward the TCP segments to a specified host and port. This option is like the previous one except that it doesn't examine the contents of the TCP segment's data area to verify that it contains an HTTP request.
- Allow all incoming IP datagrams that don't belong to an established connection and forward their contents to a specified host. This is the least secure option, but it can be sufficient for some applications. For example, the specified host may be an embedded system that accepts only HTTP requests from specific IP addresses, ignoring all other communications.

Other configuration options a firewall might have include these:

- Specify remote IP addresses that a local host can receive traffic from. This option is useful if your embedded system communicates only with a specific IP address or series of IP addresses.
- Allow only specified computers to communicate with computers outside the firewall. Or block specified computers from communicating with computers outside the firewall. These options enable you to allow an

embedded system to communicate on the Internet while protecting other computers in the local network that don't need Internet access. The firewall may enable you to identify the computers by IP address or by Ethernet hardware address. Using hardware addresses can be useful if the IP addresses are assigned dynamically and are subject to change.

- Block any outgoing communication where the Source Address of the datagram isn't a local address. (A firewall with NAT support will translate the local address to the firewall's public IP address when sending the datagram on the Internet.) This option can prevent some malicious software from using your local computers to access the Internet.
- Allow a host behind the firewall to communicate without firewall protection. The host is said to reside in a "demilitarized zone" (DMZ) and must have its own public IP address.

Embedded Firewalls

If you have a device that connects to the Internet by itself, without connecting to a local network, you may be able to provide adequate protection in the device firmware, without requiring a separate firewall device. This is especially true if the device requires only specific and limited Internet access. For example, if the device communicates with a single IP address over a specific user port, the firmware can ignore all other network communications. For other applications, requiring all users to enter a user name and password before accessing the device's resources (as in the Basic Authentication examples earlier in this chapter) may provide adequate protection.

Restrict Access with User Names and Passwords

A firewall enables you to control which local resources are available on the Internet and which IP addresses can access those resources. But firewalls filter only on the information in IP and other headers. They can't identify specific, authorized users who may be using IP addresses that the firewall doesn't know about ahead of time.

A solution is to provide authorized users with a password and to require users to enter the password before accessing a resource. For additional security and to identify who is accessing the resources, you can require a user name in addition to a password. Each user name and password combination can be unique to a user, so different users can have different access. The accepted passwords and user names may be hard-coded into the firmware, with authorized users informed of the values to use. Or you may want to allow users to obtain access to a resource by filling out a form that requires selecting a user name and password. The form can request additional information as well.

Two words you'll encounter relating to password protection are *authentication* and *authorization.* A user who wants to access a protected resource must provide authentication, or proof that the user has permission to access the resource. On receiving a valid user name and password, the server grants authorization, or permission, to access the resource.

Basic Authentication and Digest Authentication

The examples at the beginning of this chapter showed how to use Basic Authentication to require a user name and password before accessing a resource.

A more secure option than Basic Authentication is Digest Authentication. To access a resource protected with Digest Authentication, the user must provide a *message digest,* which is a 32-character ASCII hex string created from information provided by both the client requesting the resource and the server that is hosting the resource. The information that goes into creating the message digest includes a *nonce* value that the server returns in response to a request for a protected resource, a user name, a password, a realm, and the request. The default method for obtaining the message-digest string is the MD5 algorithm described in *RFC 1321: The MD5 Message-Digest Algorithm.*

The nonce value provided by the server typically incorporates a time stamp and an Etag value that identifies the resource being requested. The time stamp enables the server to allow access for a specified time before requiring re-authentication. A server can use the Etag value to prevent replay attacks, where an unauthorized user requests an updated version of a resource previously returned to an authorized user.

Rabbit Semiconductor's Dynamic C includes functions that support Digest Authentication on Web servers hosted by Rabbit modules. Some older Web browsers don't support Digest Authentication.

HTML Passwords

For very basic password protection, HTML's password box can do the job. A password box on a Web page is just like a text box except that the `TYPE` attribute of HTML's `input` tag is `"password"`:

```
<input type = "password" name=mypassword maxlength=20>
```

When a user types a password in the box, the browser displays a dot for each character typed. When the user clicks the form's **Submit** button, the browser sends the password to the server without encrypting it. Unlike Basic Authentication, which many servers support automatically, the use of this type of password box is application-specific. The server must provide program code to check the password and take appropriate action.

Additional Password Considerations

Be sure to limit access to any files that store user names and passwords so they aren't easily viewable by unauthorized users. And be aware that password protection only limits who can request a resource. The resource itself isn't encrypted when traveling on the network.

As previous chapters have shown, user names and passwords can also control access to e-mail mailboxes and files on an FTP server.

Validate User Data

Another way a device's resources can be at risk is via data received from a client, such as data submitted on a form. Users can cause harm due to malicious behavior or carelessness.

Limiting Input Range

When enabling users to enter data to be used in configuring or controlling a device, it's always a good idea to limit valid inputs to a reasonable range. For example, in a system that controls heating and cooling for a house, you may want to allow inputs only between, say, 50 and 80 degrees Fahrenheit. That way, if someone mistakenly types a thermostat setting of 0 instead of 60, the system can display an error message instead of attempting to implement the setting.

Limiting Input Length

On a form, input tags that enable users to enter text should always have a `maxlength` attribute that limits the number of characters a user can send. This line of HTML code creates an input box called temperature and allows the user to enter up to three characters:

```
<input type= "text" name="temperature" maxlength=3>
```

Limiting the length helps to ensure that the received value doesn't extend beyond the amount of memory reserved for the value on the server.

SSI Vulnerabilities

Chapter 6 introduced SSI directives. A couple of directives can have unintended consequences. The `#exec` directive can request the server to execute program code, and the `#include` directive can request the contents of a file to be included in a requested resource. If your server supports these directives but they're unneeded by applications, it's best to disable them if possible. In any case, to guard against unauthorized release of data or execution of program code, anything stored in the device that should remain private should be in an area unavailable to unauthorized users.

The Rabbit's *http.lib* library supports `#include` and an `#exec cmd` directive, which can execute a function named in an `HTTPSPEC_FUNCTION` entry in the application's `HttpSpec` structure.

Encrypt Private Data

The fourth rule for securing network resources is to encrypt data that must remain private. Basic and Digest Authentication encrypt passwords. It's also possible to encrypt any data exchanged between two computers.

Encrypting and decrypting large amounts of data can take up a lot of CPU cycles and time. On small embedded systems, the challenge is to obtain the needed security without overwhelming the system's resources. Options such as AES encryption and stand-alone firewalls that support Virtual Private Network protocols are two possible solutions for embedded systems.

AES (Rijndael) Encryption

In 1997, the U.S. National Institute of Standards and Technology (NIST) began a search for a new encryption standard that was royalty-free, easy to implement even on small embedded systems, and able to withstand attack. In 2001, Federal Information Processing Standard (FIPS) 197 designated the Rijndael algorithm the winner of the search. The algorithm was named the government's Advanced Encryption Standard (AES) to use for sensitive but unclassified information. Entities other than the U.S. Government are welcome to use the algorithm as well, of course.

Rabbit Semiconductor's Dynamic C offers a library module with support for the Rijndael Advanced Encryption Standard (AES) cipher.

Virtual Private Networks

Another option for securing network data is a virtual private network (VPN). The computers at each end of the VPN can use authentication and encryption to ensure that the data is secure from spying and to block all other traffic from entering the VPN.

The program code required to implement a VPN can be too complex and time-consuming to develop for a small embedded system. However, just about any system can communicate over a VPN by connecting to a relatively inexpensive firewall device with VPN support.

VPNs use IP Security (IPsec) protocols for encryption and authentication. A variety of RFC documents cover the protocols. A good place to start is with *RFC 2411: IP Security Document Roadmap*.

To establish a VPN, a computer at each end of the network must have software that knows how to use the required protocols to establish a connection to the other end. Windows XP includes an IPSec Security Manager that enables PCs to communicate over a VPN. For embedded systems, the easiest way to support VPN is to connect the system to a firewall device that supports VPNs.

Firewalls that support VPNs typically include a variety of configuration options. At the local network, you can enable a single IP address, an entire subnet, or a user-specified range of addresses within a subnet to access the VPN. You can specify that the local network will accept VPN communications from a specified IP address or domain name, or from any requesting host.

To use encryption, both ends of the VPN must agree on the type of encryption to use and they must share a key that enables each end to encrypt and decrypt network traffic. Encryption options include AES and the older methods 3DES and DES. Authentication options include MD5 and the more secure 160-bit Secure Hash Algorithm (SHA).

When both ends have been configured, the devices can communicate and attempt to establish the VPN. When the VPN has been established, the two devices can use encryption to transfer data securely.

Secure Sockets Layer Encryption

Many Web browsers support the Secure Sockets Layer (SSL) protocol for encrypting data such as the credit-card numbers customers send to on-line retailers. SSL uses public-key cryptography, which uses separate keys for encrypting and decrypting. The computer requesting the encrypted data generates a public key for encrypting and a private key for decrypting. The sender of the data uses the public key in encrypting the data. Decrypting requires the private key, which only the receiving computer has access to.

SSL encryption is very secure but requires more resources than many small embedded systems can provide. Netburner is one company that offers SSL support for its products, which use Motorola's 32-bit ColdFire processors.

Index

WIRELESS EMBEDDED NETWORKING:

Wireless sensor networks
Introduction
Applications
Network Topology
Localization
Time Synchronization
Energy efficient MAC protocols
SMAC
Energy efficient and robust routing
Data Centric routing

Introduction

1.1 Wireless sensor networks: the vision

Recent technological advances allow us to envision a future where large numbers of low-power, inexpensive sensor devices are densely embedded in the physical environment, operating together in a wireless network. The envisioned applications of these wireless sensor networks range widely: ecological habitat monitoring, structure health monitoring, environmental contaminant detection, industrial process control, and military target tracking, among others.

A US National Research Council report titled *Embedded Everywhere* notes that the use of such networks throughout society "could well dwarf previous milestones in the information revolution" [47]. Wireless sensor networks provide bridges between the virtual world of information technology and the real physical world. They represent a fundamental paradigm shift from traditional inter-human personal communications to autonomous inter-device communications. They promise unprecedented new abilities to observe and understand large-scale, real-world phenomena at a fine spatio-temporal resolution. As a result, wireless sensor networks also have the potential to engender new breakthrough scientific advances.

While the notion of networking distributed sensors and their use in military and industrial applications dates back at least to the 1970s, the early systems were primarily wired and small in scale. It was only in the 1990s – when wireless technologies and low-power VLSI design became feasible – that researchers began envisioning and investigating large-scale embedded wireless sensor networks for dense sensing applications.

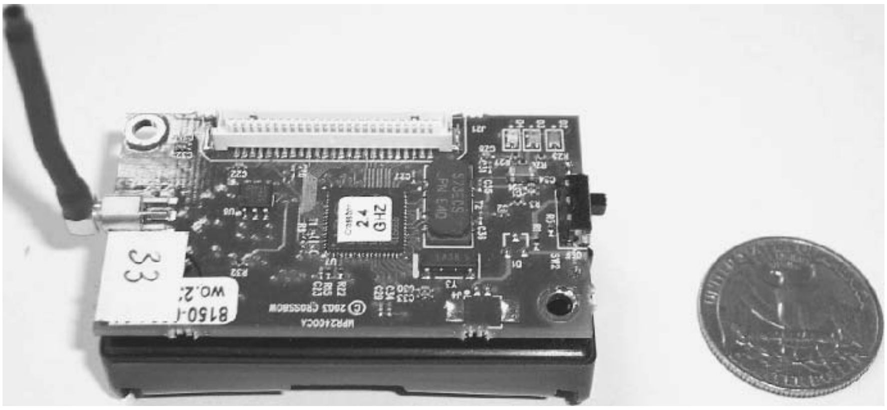


Figure 1.1 A Berkeley mote (MICAz MPR2400 series)

Perhaps one of the earliest research efforts in this direction was the low-power wireless integrated microsensors (LWIM) project at UCLA funded by DARPA [98]. The LWIM project focused on developing devices with low-power electronics in order to enable large, dense wireless sensor networks. This project was succeeded by the Wireless Integrated Networked Sensors (WINS) project a few years later, in which researchers at UCLA collaborated with Rockwell Science Center to develop some of the first wireless sensor devices. Other early projects in this area, starting around 1999–2000, were also primarily in academia, at several places including MIT, Berkeley, and USC.

Researchers at Berkeley developed embedded wireless sensor networking devices called motes, which were made publicly available commercially, along with TinyOS, an associated embedded operating system that facilitates the use of these devices [81]. Figure 1.1 shows an image of a Berkeley mote device. The availability of these devices as an easily programmable, fully functional, relatively inexpensive platform for experimentation, and real deployment has played a significant role in the ongoing wireless sensor networks revolution.

1.2 Networked wireless sensor devices

As shown in Figure 1.2, there are several key components that make up a typical wireless sensor network (WSN) device:

1. **Low-power embedded processor:** The computational tasks on a WSN device include the processing of both locally sensed information as well as information communicated by other sensors. At present, primarily due to economic

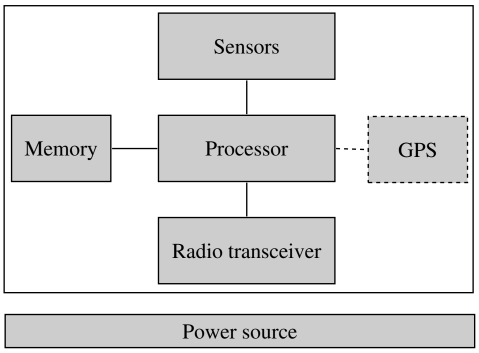


Figure 1.2 Schematic of a basic wireless sensor network device

constraints, the embedded processors are often significantly constrained in terms of computational power (e.g., many of the devices used currently in research and development have only an eight-bit 16-MHz processor). Due to the constraints of such processors, devices typically run specialized component-based embedded operating systems, such as TinyOS. However, it should be kept in mind that a sensor network may be heterogeneous and include at least some nodes with significantly greater computational power. Moreover, given Moore's law, future WSN devices may possess extremely powerful embedded processors. They will also incorporate advanced low-power design techniques, such as efficient sleep modes and dynamic voltage scaling to provide significant energy savings.

2. **Memory/storage:** Storage in the form of random access and read-only memory includes both program memory (from which instructions are executed by the processor), and data memory (for storing raw and processed sensor measurements and other local information). The quantities of memory and storage on board a WSN device are often limited primarily by economic considerations, and are also likely to improve over time.
3. **Radio transceiver:** WSN devices include a low-rate, short-range wireless radio (10–100 kbps, $<$100 m). While currently quite limited in capability too, these radios are likely to improve in sophistication over time – including improvements in cost, spectral efficiency, tunability, and immunity to noise, fading, and interference. Radio communication is often the most power-intensive operation in a WSN device, and hence the radio must incorporate energy-efficient sleep and wake-up modes.
4. **Sensors:** Due to bandwidth and power constraints, WSN devices primarily support only low-data-rate sensing. Many applications call for multi-modal sensing, so each device may have several sensors on board. The specific

sensors used are highly dependent on the application; for example, they may include temperature sensors, light sensors, humidity sensors, pressure sensors, accelerometers, magnetometers, chemical sensors, acoustic sensors, or even low-resolution imagers.

5. **Geopositioning system:** In many WSN applications, it is important for all sensor measurements to be location stamped. The simplest way to obtain positioning is to pre-configure sensor locations at deployment, but this may only be feasible in limited deployments. Particularly for outdoor operations, when the network is deployed in an *ad hoc* manner, such information is most easily obtained via satellite-based GPS. However, even in such applications, only a fraction of the nodes may be equipped with GPS capability, due to environmental and economic constraints. In this case, other nodes must obtain their locations indirectly through network localization algorithms.
6. **Power source:** For flexible deployment the WSN device is likely to be battery powered (e.g. using LiMH AA batteries). While some of the nodes may be wired to a continuous power source in some applications, and energy harvesting techniques may provide a degree of energy renewal in some cases, the finite battery energy is likely to be the most critical resource bottleneck in most WSN applications.

Depending on the application, WSN devices can be networked together in a number of ways. In basic data-gathering applications, for instance, there is a node referred to as the *sink* to which all data from *source* sensor nodes are directed. The simplest logical topology for communication of gathered data is a single-hop star topology, where all nodes send their data directly to the sink. In networks with lower transmit power settings or where nodes are deployed over a large area, a multi-hop tree structure may be used for data-gathering. In this case, some nodes may act both as sources themselves, as well as routers for other sources.

One interesting characteristic of wireless sensor networks is that they often allow for the possibility of intelligent in-network processing. Intermediate nodes along the path do not act merely as packet forwarders, but may also examine and process the content of the packets going through them. This is often done for the purpose of data compression or for signal processing to improve the quality of the collected information.

1.3 Applications of wireless sensor networks

The several envisioned applications of WSN are still very much under active research and development, in both academia and industry. We describe a few applications from different domains briefly to give a sense of the wide-ranging scope of this field:

1. **Ecological habitat monitoring:** Scientific studies of ecological habitats (animals, plants, micro-organisms) are traditionally conducted through hands-on field activities by the investigators. One serious concern in these studies is what is sometimes referred to as the "observer effect" – the very presence and potentially intrusive activities of the field investigators may affect the behavior of the organisms in the monitored habitat and thus bias the observed results. Unattended wireless sensor networks promise a cleaner, remote-observer approach to habitat monitoring. Further, sensor networks, due to their potentially large scale and high spatio-temporal density, can provide experimental data of an unprecedented richness.

   One of the earliest experimental deployments of wireless sensor networks was for habitat monitoring, on Great Duck Island, Maine [130]. A team of researchers from the Intel Research Lab at Berkeley, University of California at Berkeley, and the College of the Atlantic in Bar Harbor deployed wireless sensor nodes in and around burrows of Leach's storm petrel, a bird which forms a large colony on that island during the breeding season. The sensor-network-transmitted data were made available over the web, via a base station on the island connected to a satellite communication link.

2. **Military surveillance and target tracking:** As with many other information technologies, wireless sensor networks originated primarily in military-related research. Unattended sensor networks are envisioned as the key ingredient in moving towards network-centric warfare systems. They can be rapidly deployed for surveillance and used to provide battlefield intelligence regarding the location, numbers, movement, and identity of troops and vehicles, and for detection of chemical, biological, and nuclear weapons.

   Much of the impetus for the fast-growing research and development of wireless sensor networks has been provided though several programs funded by the US Defense Advanced Research Projects Agency (DARPA), most notably through a program known as Sensor Information Technology (SensIT) [188] from 1999 to 2002. Indeed, many of the leading US researchers and entrepreneurs in the area of wireless sensor networks today have been and are being funded by these DARPA programs.

3. **Structural and seismic monitoring:** Another class of applications for sensor networks pertains to monitoring the condition of civil structures [231]. The structures could be buildings, bridges, and roads; even aircraft. At present the health of such structures is monitored primarily through manual and visual

inspections or occasionally through expensive and time-consuming technologies, such as X-rays and ultrasound. Unattended networked sensing techniques can automate the process, providing rich and timely information about incipient cracks or about other structural damage. Researchers envision deploying these sensors densely on the structure – either literally embedded into the building material such as concrete, or on the surface. Such sensor networks have potential for monitoring the long-term wear of structures as well as their condition after destructive events, such as earthquakes or explosions. A particularly compelling futuristic vision for the use of sensor networks involves the development of controllable structures, which contain actuators that react to real-time sensor information to perform “echo-cancellation" on seismic waves so that the structure is unaffected by any external disturbance.

4. **Industrial and commercial networked sensing:** In industrial manufacturing facilities, sensors and actuators are used for process monitoring and control. For example, in a multi-stage chemical processing plant there may be sensors placed at different points in the process in order to monitor the temperature, chemical concentration, pressure, etc. The information from such real-time monitoring may be used to vary process controls, such as adjusting the amount of a particular ingredient or changing the heat settings. The key advantage of creating wireless networks of sensors in these environments is that they can significantly improve both the cost and the flexibility associated with installing, maintaining, and upgrading wired systems [131]. As an indication of the commercial promise of wireless embedded networks, it should be noted that there are already several companies developing and marketing these products, and there is a clear ongoing drive to develop related technology standards, such as the IEEE 802.15.4 standard [94], and collaborative industry efforts such as the Zigbee Alliance [244].

1.4 Key design challenges

Wireless sensor networks are interesting from an engineering perspective, because they present a number of serious challenges that cannot be adequately addressed by existing technologies:

1. **Extended lifetime:** As mentioned above, WSN nodes will generally be severely energy constrained due to the limitations of batteries. A typical alkaline battery, for example, provides about 50 watt-hours of energy; this may translate to less than a month of continuous operation for each node in full active mode. Given the expense and potential infeasibility of monitoring and

device. Additional post-deployment self-configuration mechanisms are therefore required to obtain the desired coverage and connectivity. In case of a uniform random deployment, the only parameters that can be controlled *a priori* are the numbers of nodes and some related settings on these nodes, such as their transmission range. We shall discuss some results from Random Graph Theory in Section 2.4 that provide useful insights into the settings of these parameters.

Regardless of whether the deployment is randomized or structured, the connectivity properties of the network topology can be further adjusted after deployment by varying transmit powers. We will discuss variable power-based topology control techniques in Section 2.5.

2.3 Network topology

The communication network can be configured into several different topologies, as seen in Figure 2.1. We describe these topologies below.

2.3.1 Single-hop star

The simplest WSN topology is the single-hop star shown in Figure 2.1(a). Every node in this topology communicates its measurements directly to the gateway. Wherever feasible, this approach can significantly simplify design, as the networking concerns are reduced to a minimum. However, the limitation of this topology is its poor scalability and robustness properties. For instance, in larger areas, nodes that are distant from the gateway will have poor-quality wireless links.

2.3.2 Multi-hop mesh and grid

For larger areas and networks, multi-hop routing is necessary. Depending on how they are placed, the nodes could form an arbitrary mesh graph as in Figure 2.1(b) or they could form a more structured communication graph such as the 2D grid structure shown in Figure 2.1(c).

2.3.3 Two-tier hierarchical cluster

Perhaps the most compelling architecture for WSN is a deployment architecture where multiple nodes within each local region report to different clusterheads [76]. There are a number of ways in which such a hierarchical architecture

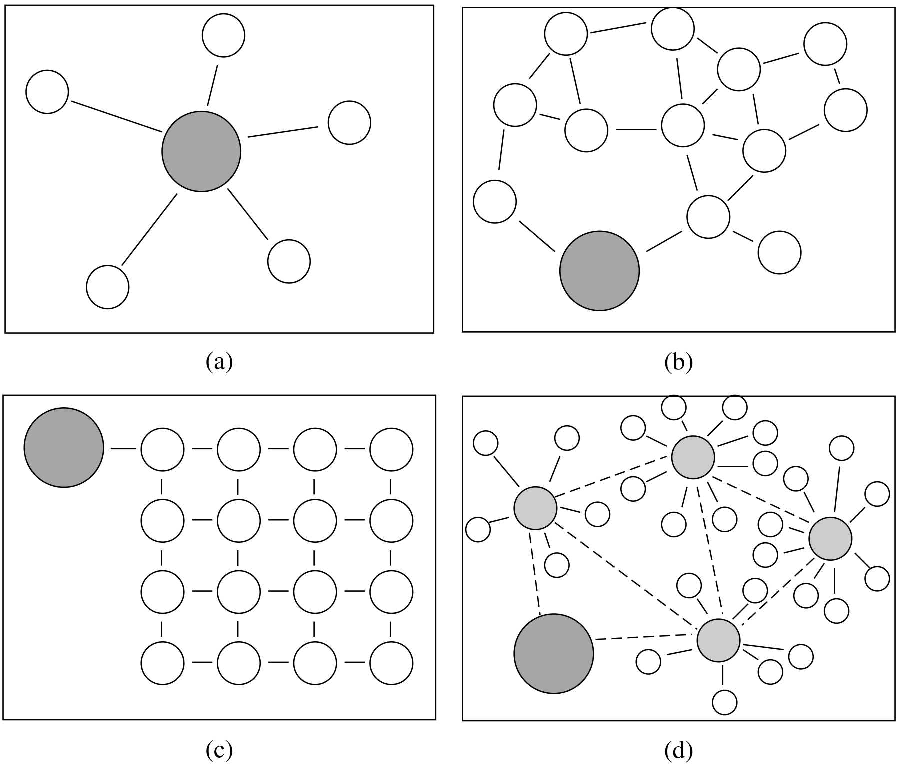


Figure 2.1 Different deployment topologies: (a) a star-connected single-hop topology, (b) flat multi-hop mesh, (c) structured grid, and (d) two-tier hierarchical cluster topology

may be implemented. This approach becomes particularly attractive in heterogeneous settings when the cluster-head nodes are more powerful in terms of computation/communication [90, 114]. The advantage of the hierarchical cluster-based approach is that it naturally decomposes a large network into separate zones within which data processing and aggregation can be performed locally. Within each cluster there could be either single-hop or multi-hop communication. Once data reach a cluster-head they would then be routed through the second-tier network formed by cluster-heads to another cluster-head or a gateway. The second-tier network may utilize a higher bandwidth radio or it could even be a wired network if the second-tier nodes can all be connected to the wired infrastructure. Having a wired network for the second tier is relatively easy in building-like environments, but not for random deployments in remote locations. In random deployments there may be no designated cluster-heads; these may have to be determined by some process of self-election.

Localization

3.1 Overview

Wireless sensor networks are fundamentally intended to provide information about the spatio-temporal characteristics of the observed physical world. Each individual sensor observation can be characterized essentially as a tuple of the form $< S, T, M >$, where S is the spatial location of the measurement, T the time of the measurement, and M the measurement itself. We shall address the following fundamental question in this chapter: How can the spatial location of nodes be determined?

The location information of nodes in the network is fundamental for a number of reasons:

1. **To provide location stamps** for individual sensor measurements that are being gathered.
2. **To locate and track point objects** in the environment.
3. **To monitor the spatial evolution of a diffuse phenomenon** over time, such as an expanding chemical plume. For instance, this information is necessary for in-network processing algorithms that determine and track the changing boundaries of such a phenomenon.
4. **To determine the quality of coverage**. If node locations are known, the network can keep track of the extent of spatial coverage provided by active sensors at any time.
5. **To achieve load balancing** in topology control mechanisms. If nodes are densely deployed, geographic information of nodes can be used to selectively shut down some percentage of nodes in each geographic area to conserve energy, and rotate these over time to achieve load balancing.

6. **To form clusters**. Location information can be used to define a partition of the network into separate clusters for hierarchical routing and collaborative processing.
7. **To facilitate routing** of information through the network. There are a number of geographic routing algorithms that utilize location information instead of node addresses to provide efficient routing.
8. **To perform efficient spatial querying**. A sink or gateway node can issue queries for information about specific locations or geographic regions. Location information can be used to scope the query propagation instead of flooding the whole network, which would be wasteful of energy.

We should, at the outset, make it clear that localization may not be a significant challenge in all WSN. In structured, carefully deployed WSN (for instance in industrial settings, or scientific experiments), the location of each sensor may be recorded and mapped to a node ID at deployment time. In other contexts, it may be possible to obtain location information using existing infrastructure, such as the satellite-based GPS [141] or cellular phone positioning techniques [218].

However, these are not satisfactory solutions to all contexts. A-priori knowledge of sensor locations will not be available in large-scale and ad hoc deployments. A pure-GPS solution is viable only if all nodes in the network can be provided with a potentially expensive GPS receiver and if the deployed area provides good satellite coverage. Positioning using signals directly from cellular systems will not be applicable for densely deployed WSN, because they generally offer poor location accuracy (on the order of tens of meters). If only a subset of the nodes have known location a priori, the position of other nodes must still be determined through some localization technique.

3.2 Key issues

Localization is quite a broad problem domain [80, 185], and the component issues and techniques can be classified on the basis of a number of key questions.

1. **What to localize?** This refers to identifying which nodes have a priori known locations (called reference nodes) and which nodes do not (called unknown nodes). There are a number of possibilities. The number and fraction of reference nodes in a network of n nodes may vary all the way from 0 to $n-1$. The reference nodes could be static or mobile; as could

the unknown nodes. The unknown nodes may be cooperative (e.g. participants in the network, or robots traversing the networked area) or non-cooperative (e.g. targets being surveilled). The last distinction is important because non-cooperative nodes cannot participate actively in the localization algorithm.

2. **When to localize?** In most cases, the location information is needed for all unknown nodes at the very beginning of network operation. In static environments, network localization may thus be a one-shot process. In other cases, it may be necessary to provide localization on-the-fly, or refresh the localization process as objects and network nodes move around, or improve the localization by incorporating additional information over time. The time scales involved may vary considerably from being of the order of minutes to days, even months.
3. **How well to localize?** This pertains to the resolution of location information desired. Depending on the application, it may be required for the localization technique to provide absolute (x, y, z) coordinates, or perhaps it will suffice to provide relative coordinates (e.g. "south of node 24 and east of node 22"); or symbolic locations (e.g. "in room A", "in sector 23", "near node 21"). Even in case of absolute locations, the required accuracy may be quite different (e.g. as good as ± 20 cm or as rough as ± 10 m). The technique must provide the desired type and accuracy of localization, taking into account the available resources (such as computational resources, time-synchronization capability, etc.).
4. **Where to localize?** The actual location computation can be performed at several different points in the network: at a central location once all component information such as inter-node range estimates is collected; in a distributed iterative manner within reference nodes in the network; or in a distributed manner within unknown nodes. The choice may be determined by several factors: the resource constraints on various nodes, whether the node being localized is cooperative, the localization technique employed, and, finally, security considerations.
5. **How to localize?** Finally, different signal measurements can be used as inputs to different localization techniques. The signals used can vary from narrowband radio signal strength readings or packet-loss statistics, UWB RF signals, acoustic/ultrasound signals, infrared. The signals may be emitted and measured by the reference nodes, by the unknown nodes, or both. The basic localization algorithm may be based on a number of techniques, such as proximity, calculation of centroids, constraints, ranging, angulation, pattern recognition, multi-dimensional scaling, and potential methods.

3.3 Localization approaches

Generally speaking, there are two approaches to localization:

1. **Coarse-grained localization using minimal information:** These typically use a small set of discrete measurements, such as the information used to compute location. Minimal information could include binary proximity (can two nodes hear each other or not?), near–far information (which of two nodes is closer to a given third node?), or cardinal direction information (is one node in the north, east, west, or south sector of the other given node?).
2. **Fine-grained localization using detailed information:** These are typically based on measurements, such as RF power, signal waveform, time stamps, etc., that are either real-valued or discrete with a large number of quantization levels. These include techniques based on radio signal strengths, timing information, and angulation.

The tradeoff that emerges between the two approaches is easy to see: while minimal information techniques are simpler to implement, and likely involve lower resource consumption and equipment costs, they provide lower accuracy than the detailed information techniques. We shall now describe specific techniques in detail.

We shall start first with the *node localization* problem involving a single unknown node and several reference nodes, and then discuss the problem of *network localization* where there are several unknown nodes in a multi-hop network.

3.4 Coarse-grained node localization using minimal information

3.4.1 Binary proximity

Perhaps the most basic location technique is that of binary proximity – involving a simple decision of whether two nodes are within reception range of each other. A set of references nodes are placed in the environment in some non-overlapping (or nearly non-overlapping) manner. Either the reference nodes periodically emit beacons, or the unknown node transmits a beacon when it needs to be localized. If reference nodes emit beacons, these include their location IDs. The unknown node must then determine which node it is closest to, and this provides a coarse-grained localization. Alternatively, if the unknown node emits a beacon, the reference node that hears the beacon uses its own location to determine the location of the unknown node.

An excellent example of proximity detection as a means for localization is the Active Badge location system [221] meant for an indoor office environment. This system consists of small badge cards (about 5 square centimeters in size and less than a centimeter thick) sending unique beacon signals once every 15 seconds with a 6 meter range. The active badges, in conjunction with a wired sensor network that provides coverage throughout a building, provide room-level location resolution. A much larger application of localization, using binary proximity detection, is with passive radio frequency identification (RFID) tags, which can be detected by readers within a similar short range [52]. Today there are a large number of inventory-tracking applications envisioned for RFIDs. A key difference in RFID proximity detection compared with active badges is that the unknown nodes are passive tags, being queried by the reference nodes in the sensor network. These examples show that even the simplest localization technique can be of considerable use in practice.

3.4.2 Centroid calculation

The same proximity information can be used to greater advantage when the density of reference nodes is sufficiently high that there are several reference nodes within the range of the unknown node. Consider a two-dimensional scenario. Let there be n reference nodes detected within the proximity of the unknown node, with the location of the ith such reference denoted by (x_i, y_i). Then, in this technique, the location of the unknown node (x_u, y_u) is determined as

$$x_u = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$y_u = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{3.1}$$

This simple centroid technique has been investigated using a model with each node having a simple circular range R in an infinite square mesh of reference nodes spaced a distance d apart [16]. It is shown through simulations that, as the overlap ratio R/d is increased from 1 to 4, the average RMS error in localization is reduced from $0.5d$ to $0.25d$.

3.4.3 Geometric constraints

If the bounds on radio or other signal coverage for a given node can be described by a geometric shape, this can be used to provide location estimates by determining which geometric regions that node is constrained to be in, because of intersections between overlapping coverage regions.

For instance, the region of radio coverage may be upper-bounded by a circle of radius $R_{\max}$. In other words, if node B hears node A, it knows that it must be no more than a distance $R_{\max}$ from A. Now, if an unknown node hears from several reference nodes, it can determine that it must lie in the geometric region described by the intersection of circles of radius $R_{\max}$ centered on these nodes. This can be extended to other scenarios. For instance when both lower $R_{\min}$ and upper bounds $R_{\max}$ can be determined, based on the received signal strength, the shape for a single node's coverage is an annulus; when an angular sector $(\theta_{\min}, \theta_{\max})$ and a maximum range $R_{\max}$ can be determined, the shape for a single node's coverage would be a cone with given angle and radius.

Although arbitrary shapes can be potentially computed in this manner, a computational simplification that can be used to determine this bounded region is to use rectangular bounding boxes as location estimates. Thus the unknown node determines bounds $x_{\min}, y_{\min}, x_{\max}, y_{\max}$ on its position.

Figure 3.1 illustrates the use of intersecting geometric constraints for localization. Localization techniques using such geometric regions were first described by Doherty *et al.* [40]. One of the nice features of these techniques is that not only can the unknown nodes use the centroid of the overlapping region as a specific location estimate if necessary, but they can also determine a bound on the location error using the size of this region.

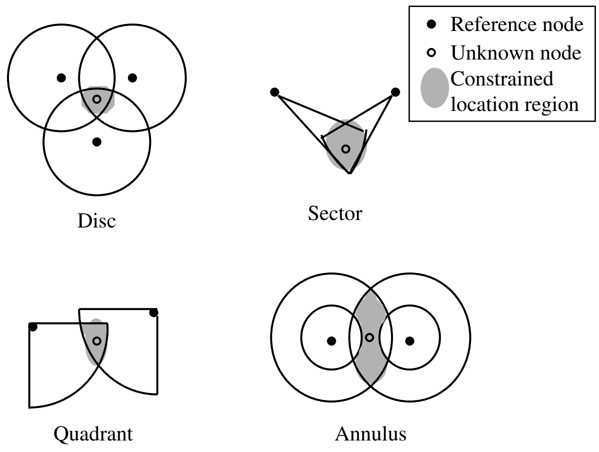


Figure 3.1 Localization using intersection of geometric constraints

When the upper bounds on these regions are tight, the accuracy of this geometric approach can be further enhanced by incorporating “negative information” about which reference nodes are *not* within range [54].

3.4.4 Approximate point in triangle (APIT)

A related approach to localization using geometric constraints is the approximate point-in-triangle (APIT) technique [72]. APIT is similar to the above techniques in that it provides location estimates as the centroid of an intersection of regions. Its novelty lies in how the regions are defined – as triangles between different sets of three reference nodes (rather than the coverage of a single node). This is illustrated in Figure 3.2. It turns out that an exact determination of whether an unknown node lies within the triangle formed by three reference nodes is impossible if nodes are static because wireless signal propagation is non-ideal. An approximate solution can be determined using near–far information [72], i.e. the ability to determine which of two nodes is nearer a third node based on signal reception. One caveat for the APIT technique is that it can provide erroneous results, because the determination of whether a node lies within a particular triangle requires quite a high density of nodes in order to provide good location accuracy.

3.4.5 Identifying codes

There is another interesting technique that utilizes overlapping coverage regions to provide localization. In this technique, referred to as the identifying code

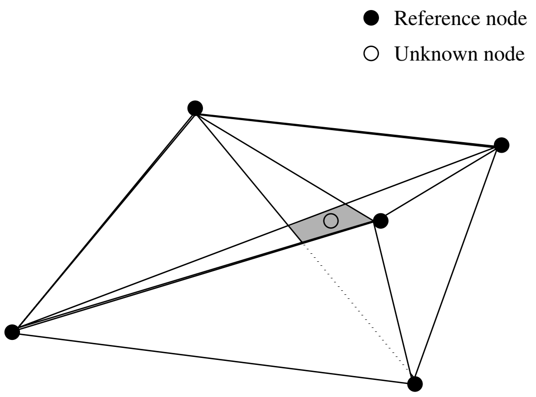


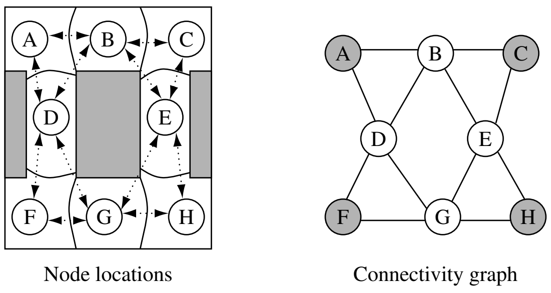
Figure 3.2 The approximate point-in-triange (APIT) technique

construction (ID-CODE) algorithm [173], the sensor deployment is planned in such a way as to ensure that each resolvable location is covered by a unique set of sensors.

The algorithm runs on a deployment region graph $G = (V, E)$ in which vertices V represent the different regions, and the edges E represent radio connectivity between regions. Let $B(v)$ be the set of vertices that are adjacent to v, together with v itself. A set of vertices $C \in V$ is referred to as an *identifying code*, if, for all $u, v \in V$, $B(v) \cap C \neq B(u) \cap C$.

It can be shown that a graph is distinguishable, i.e. there exists an identifying code for it, if and only if there are no two vertices u, v such that $B(u) = B(v)$. The goal of the algorithm is to construct an identifying code for any distinguishable graph, with each vertex in the code corresponding to a region where a reference node must be placed. Once this is done, by the definition of the identifying code, each location region in the graph will be covered by a unique set of reference nodes. This is illustrated in Figure 3.3.

While the entire set of vertices V itself is an identifying code, such a placement of a reference node in each region would clearly be inefficient. On the other hand, obtaining a minimal cardinality identifying code is known to be NP-complete. The algorithm ID-CODE is a polynomial greedy heuristic that provides good solutions in practice. There also exists a robust variant of this algorithm called r-ID-CODE [173] that can provide robust identification, i.e. guaranteeing a unique set of IDs for each location, even if there is addition or deletion of up to r ID values.



Transmitters A, F, C, H provide unique IDs for all node locations

V:	A	B	C	D	E	F	G	H
ID:	A	A,C	C	A,F	C,H	F	F,H	H

Figure 3.3 Illustration of the ID-CODE technique showing uniquely identifiable regions

3.5 Fine-grained node localization using detailed information

We now examine techniques based on detailed information. These include triangulation using distance estimates, pattern matching, and sequence decoding. Although used in the large-scale GPS, basic time-of-flight techniques using RF signals are not capable of providing precise distance estimates over short ranges typical of WSN because of synchronization limitations. Therefore other techniques such as radio signal strength (RSS) measurements and time difference of arrival (TDoA) must be used for distance-estimation.

3.5.1 Radio signal-based distance-estimation (RSS)

To a first-order approximation, mean radio signal strengths diminish with distance according to a power law. One model that is used for wireless radio propagation is the following [171]:

$$P_{\mathrm{r,dB}}(d) = P_{\mathrm{r,dB}}(d_0) - \eta 10 \log\left(\frac{d}{d_0}\right) + X_{\sigma,\mathrm{dB}} \tag{3.2}$$

where $P_{\mathrm{r,dB}}(d)$ is the received power at distance d and $P(d_0)$ is the received power at some reference distance d_0, η the path-loss exponent, and $X_{\sigma,\mathrm{dB}}$ a log-normal random variable with variance σ^2 that accounts for fading effects. So, in theory, if the path-loss exponent for a given environment is known the received signal strength can be used to estimate the distance. However, the fading term often has a large variance, which can significantly impact the quality of the range estimates. This is the reason RF-RSS-based ranging techniques may offer location accuracy only on the order of meters or more [154]. RSS-based ranging may perform much better in situations where the fading effects can be combatted by diversity techniques that take advantage of separate spatio-temporally correlated signal samples.

3.5.2 Distance-estimation using time differences (TDoA)

As we have seen, time-of-flight techniques show poor performance due to precision constraints, and RSS techniques, although somewhat better, are still limited by fading effects. A more promising technique is the combined use of ultrasound/acoustic and radio signals to estimate distances by determining the TDoA of these signals [164, 183, 223]. This technique is conceptually quite simple, and is illustrated in Figure 3.4. The idea is to simultaneously transmit both the radio and acoustic signals (audible or ultrasound) and measure the times T_r and T_s of the

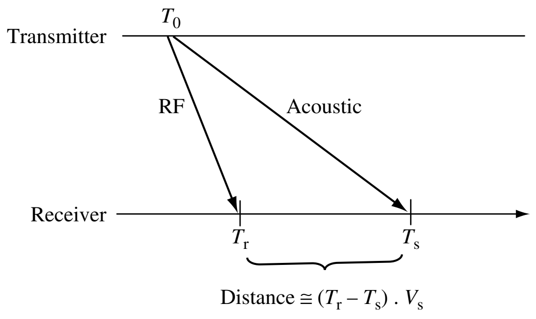


Figure 3.4 Ranging based on time difference of arrival

arrival of these signals respectively at the receiver. Since the speed of the radio signal is much larger than the speed of the acoustic signal, the distance is then simply estimated as $(T_s - T_r) \cdot V_s$, where V_s is the speed of the acoustic signal.

One minor limitation of acoustic ranging is that it generally requires the nodes to be in fairly close proximity to each other (within a few meters) and preferably in line of sight. There is also some uncertainty in the calculation because the speed of sound varies depending on many factors such as altitude, humididity, and air temperature. Acoustic signals also show multi-path propagation effects that may impact the accuracy of signal detection. These can be mitigated to a large extent using simple spread-spectrum techniques, such as those described in [61]. The basic idea is to send a pseudo-random noise sequence as the acoustic signal and use a matched filter for detection, (instead of using a simple chirp and threshold detection).

On the whole, acoustic TDoA ranging techniques can be very accurate in practical settings. For instance, it is claimed in [183] that distance can be estimated to within a few centimeters for node separations under 3 meters. Of course, the tradeoff is that sensor nodes must be equipped with acoustic transceivers in addition to RF transceivers.

3.5.3 Triangulation using distance estimates

The location of the unknown node (x_0, y_0) can be determined based on measured distance estimates $\hat{d}_i$ to n reference nodes $\{(x_1, y_1), \ldots, (x_i, y_i), \ldots, (x_n, y_n)\}$. This can be formulated as a least squares minimization problem.

Let d_i be the correct Euclidean distance to the n reference nodes, i.e.:

$$d_i = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} \tag{3.3}$$

Thus the difference between the measured and actual distances can be represented as

$$\rho_i = \hat{d}_i - d_i \tag{3.4}$$

The least squares minimization problem is then to determine the (x_0, y_0) that minimizes $\sum_{i=1}^{n}(\rho_i)^2$. This problem can be solved by the use of gradient descent techniques or by iterative successive approximation techniques such as described in [146]. An alternative is the following approach, which provides a numerical solution to an over-determined $(n \geq 3)$ linear system [183].

The over-determined linear system can be obtained as follows. Rearranging and squaring terms in equation (3.3), we would have n such equations:

$$x_i^2 + y_i^2 - d_i^2 = 2x_0x_i + 2y_0y_i - (x_0^2 + y_0^2) \tag{3.5}$$

By subtracting out the nth equation from the rest, we would have $n-1$ equations of the following form:

$$x_i^2 + y_i^2 - x_n^2 - y_n^2 - d_i^2 + d_n^2 = x_0 2(x_i - x_n) + y_0 2(y_i - y_n) \tag{3.6}$$

which yields the linear relationship

$$\mathbf{A}\overline{x} = \mathbf{B} \tag{3.7}$$

where $\mathbf{A}$ is an $(n-1) \times 2$ matrix, such that the ith row of $\mathbf{A}$ is $[2(x_i - x_n) \quad 2(y_i - y_n)]$, $\overline{x}$ is the column vector representing the coordinates of the unknown location $[x_0 \quad y_0]^T$, and $\mathbf{B}$ is the $(n-1)$ element column vector whose ith term is the expression $x_i^2 + y_i^2 - x_n^2 - y_n^2 - d_i^2 + d_n^2$. Now, in practice we cannot determine $\mathbf{B}$, since we have access to only the estimated distances, so we can calculate instead the elements of the related vector $\hat{\mathbf{B}}$, which is the same as $\mathbf{B}$ with $\hat{d}_i$ substituted for d_i. Now the least squares solution to equation (3.7) is to determine an estimate for $\overline{x}$ that minimizes $\|\mathbf{A}\overline{x} - \hat{\mathbf{B}}\|_2$. Such an estimate is provided by

$$\overline{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\hat{\mathbf{B}} \tag{3.8}$$

Solving for the above may not directly yield a numerical solution if the matrix $\mathbf{A}$ is ill-conditioned, so a recommended approach is to instead use the *pseudo-inverse* $\mathbf{A}^+$ of the matrix $\mathbf{A}$:

$$\overline{x} = \mathbf{A}^+\hat{\mathbf{B}} \tag{3.9}$$

3.5.4 Angle of arrival (AoA)

Another possibility for localization is the use of angular estimates instead of distance estimates. Angles can potentially be estimated by using rotating directional beacons, or by using nodes equipped with a phased array of RF or ultrasonic receivers. A very simple localization technique, involving three rotating reference beacons at the boundary of a sensor network providing localization for all interior nodes, is described in [143]. A more detailed description of AoA-based triangulation techniques is provided in [147].

Angulation with ranging is a particularly powerful combination [27]. In theory, if the angular information provided to a given reference node can be combined with a good distance estimate to that reference node, then localization can be performed with a single reference using polar coordinate transformation. While the accuracy and precision with which angles in real systems can be determined are unclear, significant improvements can be obtained by combining accurate ranging estimates with even coarse-grained angle estimates.

3.5.5 Pattern matching (RADAR)

An alternative to measuring distances or angles that is possible in some contexts is to use a pre-determined "map" of signal coverage in different locations of the environment, and use this map to determine where a particular node is located by performing pattern matching on its measurements. An example of this technique is RADAR [5]. This technique requires the prior collection of empirical measurements (or high-fidelity simulation model) of signal strength statistics (mean, variance, median) from different reference transmitters at various locations. It is also important to take into account the directional orientation of the receiving node, as this can result in significant variations. Once this information is collected, any node in the area is localized by comparing its measurements from these references to determine which location matches the received pattern best. This technique has some advantages, in particular as a pure RF technique it has the potential to perform better than the RSS-based distance-estimation and the triangulation approach we discussed before. However, the key drawback of the technique is that it is very location specific and requires intensive data collection prior to operation; also it may not be useful in settings where the radio characteristics of the environment are highly dynamic.

3.5.6 RF sequence decoding (ecolocation)

The ecolocation technique [238] uses the relative ordering of received radio signal strengths for different references as the basis for localization. It works as follows:

1. The unknown node broadcasts a localization packet.
2. Multiple references record their RSSI reading for this packet and report it to a common calculation node.
3. The multiple RSSI readings are used to determine the ordered sequence of references from highest to lowest RSSI.
4. The region is scanned for the location for which the correct ordering of references (as measured by Euclidean distances) has the "best match" to the measured sequence. This is considered the location of the unknown node.

In an ideal environment, the measured sequence would be error free, and ecolocation would return the correct location region. However, in real environments, because of multi-path fading effects, the measured sequence is likely to be corrupted with errors. Some references, which are closer than others to the true location of the unknown node, may show a lower RSSI, while others, which are farther away, may appear earlier in the sequence. Therefore the sequence must be decoded in the presence of errors. This is why a notion of "best match" is needed.

The best match is quantified by deriving the $n(n-1)/2$ pair-wise ordering constraints (e.g. reference A is closer than reference B, reference B is closer than reference C, etc.) at each location, and determining how many of these constraints are satisfied/violated in the measured sequence. The location which provides the maximum number of satisfied constraints is the best match. Simulations and experiments suggest that ecolocation can provide generally more accurate localizations compared with other RF-only schemes, including triangulation using distance estimates. Intuitively, this is because the ordered relative sequence of RSSI values at the references provides robustness to fluctuations in the absolute RSSI value.

3.6 Network-wide localization

3.6.1 Issues

So far, we have focused on the problem of *node localization*, which is that of determining the location of a single unknown node given a number of nearby references. A broader problem in sensor systems is that of *network localization*, where several unknown nodes have to be localized in a network with a few reference nodes. While the network localization problem can rarely be neatly decomposed into a number of separate node localization problems (since there may be unknown nodes that have no reference nodes within range), the node localization and ranging techniques described above do often form an integral component of solutions to network localization.

The performance of network localization depends very much on the resources and information available within the network. Several scenarios are possible: for instance there may be no reference nodes at all, so that perhaps only relative coordinates can be determined for the unknown nodes; if present, the number/density of reference nodes may vary (generally the more reference nodes there are, the lower the network localization error); there may be just a single mobile reference. Information about which nodes are within range of each other may be available; or inter-node distance estimates may be available; inter-node angle information may be available.

Some network localization approaches are centralized, in which all the available information about known nodes, and the inter-node distances or other inter-node relationships are provided to a central node, where the solution is computed. Such a centralized approach may be sufficient in moderate-sized networks, where the nodes in the network need to be localized only once, post-deployment. Other network localization approaches are distributed, often involving the iterative communication of updated location information.

There may be several ways to measure the performance of network localization. If the ground truth is available, these can range from the full distribution/histogram of location errors, to the mean location error, to the percentage of unknown nodes that can be located within a desired accuracy. Alternatively some localization approaches provide an inherent way to estimate the uncertainty associated with each node's calculated location.

3.6.2 Constraint-based approaches

Geometric constraints can often be expressed in the form of linear matrix inequalities and linear constraints [40]. This applies radial constraints (two nodes are determined to be within range R of each other), annular constraints (a node is determined to be within ranges $[R_{\min}, R_{\max}]$ of another), angular constraints (a node is determined to be within a particular angular sector of another), as well as other convex constraints. Information about a set of reference nodes together with these constraints (which provide the inter-node relationships amongst reference as well as unknown nodes) describes a feasible set of constraints for a semidefinite program. By selecting an appropriate objective function for the program, the constraining rectangle, which bounds the location for each unknown node, can be determined.

When using bounding rectangles, a distributed iterative solution can be used [54]. In this solution, at each step nodes broadcast to their neighbors their current constrained region, which is calculated based on the overheard information about their neighbors' constrained regions at the previous step. If continued for a sufficient number of iterations, or until there is no longer a significant improvement in the bounds, this can provide a solution that is near or at optimal.

Network localization can also be performed in the presence of mobile reference/target nodes [54]. If the mobile node is a reference and able to provide an accurate location beacon, then it can substantially improve localization over time, because each new observation of the moving beacon introduces additional constraints. In theory the location error can be reduced to an arbitrarily small quantity if the moving beacon is equally likely to move to any point in the network. If the mobile node is a non-cooperative target, then the distributed iterative algorithm can be extended to provide simultaneous network localization and tracking with performance that improves over time.

3.6.3 RSS-based joint estimation

If radio signal strengths can be measured between all pairs of nodes in the network that are within detection range, then a joint maximum likelihood estimation (MLE) technique can be used to determine the location of unknown nodes in a network [154]. In the joint MLE technique, first an expression is derived for the likelihood that the obtained matrix of power measurements would be received given a particular location set for all nodes; the objective is then to find the location set that maximizes this likelihood. The performance of this joint MLE technique has been verified through simulations and experiments to show that localization of the order of 2 meters is possible when there is a high density of unknown nodes, even if there are only a few reference nodes sparsely placed.

3.6.4 Iterative multilateration

The iterative multilateration technique [183] is applicable whenever inter-node distance information is available between all neighboring nodes (regardless of whether it is obtained through RSS measurements or TDoA or any other approach). The algorithm is quite simple. It applies the basic triangulation technique for node localization (see Section 3.5.3 above) in an iterative manner to determine the locations of all nodes. One begins by determining the location of an unknown node that has the most reference nodes in its neighborhood. In a distributed version, the location of any node with sufficient references in its neighborhood may be calculated as the initial step. This node is then added to the set of reference nodes and the process is repeated. Figure 3.5 shows an example of a network with one possible sequence in which unknown nodes can each compute their location so long as at least three of their neighbors have known or already computed locations.

Note that a version of iterative multilateration can also be utilized if only connectivity information is available. In such a case, a centroid calculation could be used at each iterative step by the unknown nodes, instead of using distance-based triangulation.

The iterative multilateration technique suffers from two shortcomings: first, it may not be applicable if there is no node that has sufficient ($\geq$3 for the 2D plane) reference nodes in its neighborhood; second, the use of localized unknown nodes as reference nodes can introduce substantial cumulative error in the network localization (even if the more certain high-reference neighborhood nodes are used earlier in the iterative process).

3.6.5 Collaborative multilateration

One approach to tackle the deficiency of iterative multilateration with respect to nodes with insufficient reference neighbors is the collaborative multilateration

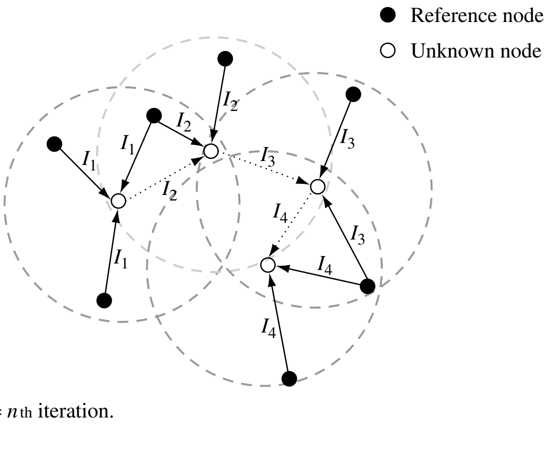


Figure 3.5 Illustration of sequence of iterative multilateration steps

approach described by Savvides *et al.* [183]. The key insight is to determine collaborative subgraphs within the network that contain reference and unknown nodes in a topology such that their positions and inter-node distances can be written as an over-constrained set of quadratic equations with a unique solution for the location of unknown nodes (which can be obtained through gradient descent or local search algorithms). Used in conjunction with iterative multilateration, this technique is generally useful in portions of the network where the reference node density is low.

3.6.6 Multi-hop distance-estimation approaches

An alternative approach to network localization utilizes estimates of distances to reference nodes that may be several hops away [146]. These distances are propagated from reference nodes to unknown nodes using a basic distance-vector technique. There are three variants of this approach:

1. DV-hop: In this approach, each unknown node determines its distance from various reference nodes by multiplying the least number of hops to the reference nodes with an estimated average distance per hop. The average distance per hop depends upon the network density, and is assumed to be known.
2. DV distance: If inter-node distance estimates are directly available for each link in the graph, then the distance-vector algorithm is used to determine the distance corresponding to the shortest distance path between the unknown nodes and reference nodes.
3. Euclidean propagation: Geometric relations can be used in addition to distance estimates to determine more accurate estimates to reference nodes. For instance consider a quadrilateral ABCR, where A and R are at opposite ends; if node A knows the distances AB, AC, BC and nodes B and C have estimates of their distance to the reference R, then A can use the geometric relations inherent in this quadrilateral to calculate an estimated distance to R.

Once distance estimates are available from each unknown node to different reference nodes throughout the network, a triangulation technique (such as described in Section 3.5.3) can be employed to determine their locations. Through simulations, it has been seen that location errors for most nodes can be kept within a typical single-hop distance [146]. In a comparative simulation study of these approaches, it has been shown that the relative performance of these three schemes depends on factors such as the radio range and accuracy of available distance estimates [113].

3.6.7 Refinement

Once a possible initial estimate for the location of unknown nodes has been determined through iterative multilateration/collaborative multilateration or the distance-vector estimation approaches, additional refinement steps can be applied [182]. Each node continues to iterate, obtaining its neighbors location estimates and using them to calculate an updated location using triangulation. After some iterations, the position updates become small and this refinement process can be stopped.

3.6.8 Force-calculation approach

An inherently distributed iterative approach to network localization is to use a physics-based analogy [86, 214]. Each node first picks a reasonable initial guess as to its location, which need not be very accurate. If $d_{i,j}$ is the calculated distance between the two nodes as per their current positions, $\hat{d}_{i,j}$ the estimated distance and $\overrightarrow{u_{i,j}}$ the unit vector between them, and H_i is the set of all neighboring nodes of i, then a vector force on a link and the resultant force on a node can be respectively defined as

$$\overrightarrow{F_{i,j}} = (d_{i,j} - \hat{d}_{i,j})\overrightarrow{u_{i,j}} \tag{3.10}$$

$$\overrightarrow{F_i} = \sum_{j \in H_i} \overrightarrow{F_{i,j}} \tag{3.11}$$

Each unknown node then updates its position in the direction of the resulting vector force in small increments over several iterations (with the force being recalculated at each step). However, it should be kept in mind that this technique may be susceptible to local minima.

3.6.9 Multi-dimensional scaling

Given a network with a sparse set of reference nodes, and a set of pair-wise distances between neighboring nodes (including reference and unknown nodes), another network localization approach utilizes a data analysis technique known as multi-dimensional scaling (MDS) [193]. It consists of the following three steps:

1. Use a distance-vector algorithm (similar to DV-distance) to generate an $n \times n$ matrix $\mathbf{M}$, whose (i, j) entry contains the estimated distance between nodes i and j.

2. Apply classical metric-MDS to determine a map that gives the locations of all nodes in relative coordinates. The classical metric MDS algorithm is a matrix-based numerical technique that solves the following least squares problem: if the estimated distance matrix $\mathbf{M}$ can be expressed as the sum of the actual distance matrix $\mathbf{D}$ and a residual error matrix $\mathbf{E}$ (i.e. $\mathbf{M} = \mathbf{D} + \mathbf{E}$), then determine possible locations for all n nodes, such that the sum of squares of the elements of $\mathbf{E}$ is minimized.
3. Take the position of reference nodes into account to obtain normalized absolute coordinates.

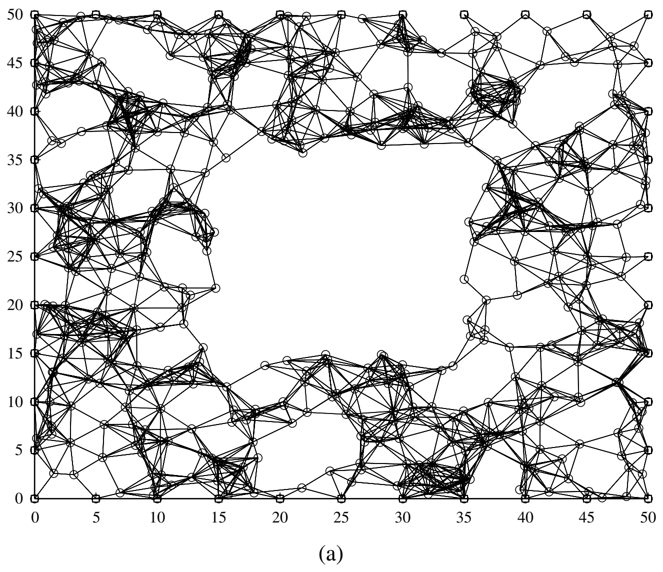
3.6.10 Reference-less localization

In some scenarios, we may encounter sensor networks that are deployed in such an *ad hoc* manner, without GPS capabilities, that there are no reference nodes whatsoever. In such a case, the best that can be hoped for is to obtain the location of the network nodes in terms of relative, instead of absolute, coordinates. While such a map is not useful for location stamping of sensor data, it can be quite useful for other functions, such as providing the information required to implement geographic routing schemes.

The multi-dimensional scaling problem (described above) can provide such a relative map, by simply eliminating step 3. Rao *et al.* [170] also develop such a technique for creating a virtual coordinate system for a network where there are no reference nodes and also where no distance estimates are available (unlike with MDS). Their algorithm is described as a progression of three scenarios with successively fewer assumptions:

1. All (and only) nodes at the boundary of the network are reference nodes.
2. Nodes at the boundary are aware that they are at the boundary, but are not reference nodes.
3. There are no reference nodes in the network, and no nodes are aware that they are at the boundary.

In the first scenario, all nodes execute a simple iterative algorithm for localization. Unknown interior nodes begin by assuming a common initial coordinate (say [0,0]), then at each step, each unknown node determines its location as the centroid of the locations of all its neighbors. It is shown that this algorithm tends to “stretch” the locations of network nodes through the location region. When the algorithm converges, nodes have determined a location that is close to their nearest boundary nodes. Figure 3.6 gives an example of a final solution.



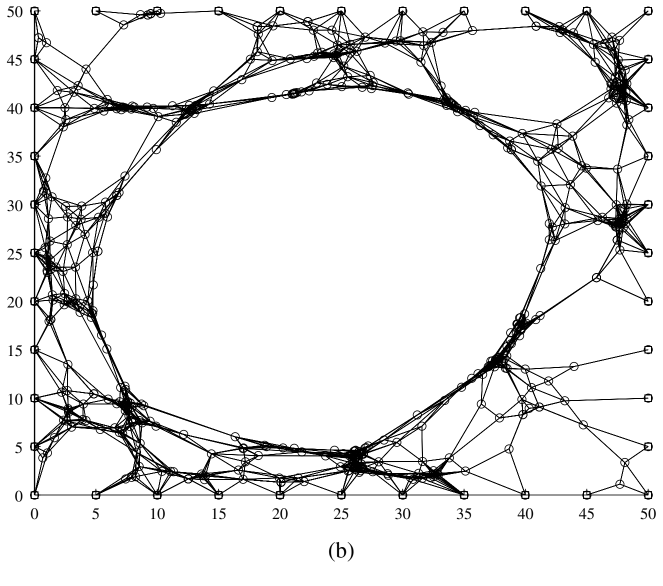


Figure 3.6 An illustration of the reference-less network localization technique assuming boundary node locations are known: (a) original map and (b) obtained relative map

While the final solution is generally not accurate, it is shown that for greedy geographic routing it results in only slightly longer routing paths and potentially even slightly better routing success rates (as non-ideal positions can sometimes improve over the local optima that arise in greedy geographic routing).

The second scenario can be reduced approximately to the first. This can be done by having the border nodes first flood messages to communicate with each other and determine the pair-wise hop-counts between themselves. These hop-counts are then used in a triangulation algorithm to obtain virtual coordinates for the set B of all border nodes by minimizing

$$\sum_{i,j \in B} (\text{hops}(i, j) - \text{dist}(i, j))^2 \tag{3.12}$$

where hops(i, j) is the number of hops between border nodes i, j, and dist(i, j) their Euclidean distance for given virtual coordinates. An additional bootstrapping mechanism ensures that all nodes calculate consistent virtual coordinates.

Finally, the third scenario can be reduced to the second. Any node that is farthest away from a common node in terms of hop-count with respect to all its two-hop neighbors can determine that it is on the border. This hop-count determination is performed through a flood from one of the bootstrap nodes.

3.7 Theoretical analysis of localization techniques

3.7.1 Cramér–Rao lower bound

One theoretical tool of utility in analyzing limitations on the performance of localization techniques is the use of the Cramér–Rao bound. The Cramér–Rao bound (CRB) is a well-known lower bound on the error variance of any unbiased estimator, and is defined as the inverse of the Fisher information matrix (a measure of information content with respect to parameters). The CRB can be derived for different assumptions about the localization technique (e.g. TOA-based, RSS-based, proximity-based, node/network localization).

The CRB has been used to investigate error performance of K-level quantized RSSI-based localization [155]. A special case is $K = 2$, which corresponds to proximity information (whether the node is within range or not). The lower bound can be improved monotonically with K, with about 50% improvement if K is large compared with just using proximity alone. On the other hand, $K = 8$ (three bits of RSS quantization) suffices to give a lower bound that is very close to the best possible. It is also found that the MLE estimator, which is a biased estimator, provides location errors with variance close to that observed with the CRB.

CRB analysis has also been used to investigate the performance of network localization under different densities, and shown to give similar trends to the iterative/collaborate multilateration technique [184]. The CRB-based analysis suggests that localization accuracy improves with network density, with diminishing returns once each node has about 6–8 neighbors on average. It also suggests, somewhat surprisingly, that increasing the fraction of beacon nodes from 4% to 20% does not dramatically decrease the localization error (under the assumptions of uniform placement, high-density, low-ranging error).

3.7.2 Unique network localization

There is a strong connection between the problem of unique network localization and a mathematical subject known as rigidity theory [46].

Definition 3

Consider a sensor network with n nodes (m reference nodes and n − m unknown nodes) located in the 2D plane, with edges between neighboring nodes. Information is available about the exact location coordinates of the reference nodes, and the exact Euclidean distance between all neighboring nodes. This network is said to be **uniquely localizable** *if there exists only one possible assignment of (x, y) coordinates to all unknown nodes that is consistent with all the available information about distances and positions.*

The key result concerning the conditions for a network to be unique localizable is the following:

Theorem 6

A network N is uniquely localizable if and only if the weighted grounded graph G'_N corresponding to it is globally rigid.

There are two terms here that need to be explained – weighted grounded graph and global rigidity. The *weighted grounded graph* G'_N is constructed from the graph described by network N (with each edge weighed by the corresponding distance) by adding additional edges between all pairs of reference nodes, labelled with the distance between them (which can be readily calculated, since reference positions are known).

We shall give an intuitive definition of global rigidity. Consider a configuration graph of points in general position on the plane, with edges connecting some of them to represent distance constraints. Is there another configuration consisting of different points on the plane that preserves all the distance constraints on the edges (excluding trivial changes, such as translations, rotations, and mirror images)? If there is not, the configuration graph is said to be globally rigid in the plane. Figure 3.7 gives examples of non-globally rigid and globally rigid configuration graphs.

There exist polynomial algorithms to determine whether a given configuration graph is globally rigid in the plane, and hence to determine if a given network is uniquely localizable. However, the problem of realizing globally rigid weighted graphs (which is closely related to actually determining possible locations of

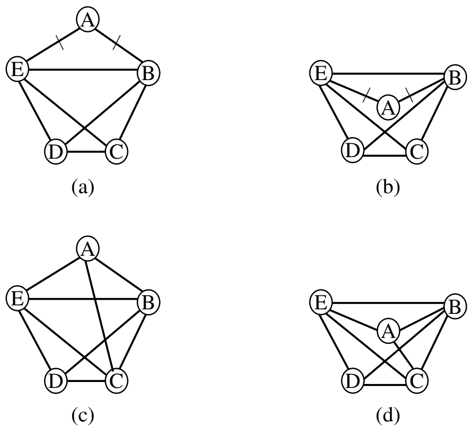


Figure 3.7 Examples of configuration graphs that are not globally rigid ((a),(b)) and that are globally rigid ((c),(d))

the unknown nodes in the corresponding network) is NP-hard. While this means that in the worst case there exist no known tractable algorithms to solve all instances, in the case of geometric random graphs, with at least three reference nodes within range of each other, there exists a critical radius threshold that is $O\left(\frac{\sqrt{\log n}}{n}\right)$, beyond which the network is uniquely localizable in polynomial time with high probability.

3.8 Summary

Determining the geographic location of nodes in a sensor network is essential for many aspects of system operation: data stamping, tracking, signal processing, querying, topology control, clustering, and routing. It is important to develop algorithms for scenarios in which only some nodes have known locations.

The design space of localization algorithms is quite large. The selection of a suitable algorithm for a given application and its performance depends upon several key factors, such as: what information about known locations is already available, whether the problem is to locate a cooperative node, how dynamic location changes are, the desired accuracy, and the constraints placed on hardware. On the basis of what needs to be localized, the location algorithms that have been proposed can be broadly classified into two categories: (i) node localization algorithms, which provide the location of a single unknown node given a number of reference nodes, and (ii) network localization algorithms, which provide the location of multiple unknown nodes in a network given other reference nodes. The node localization algorithms are often a building-block component of network localization algorithms. The accuracy of the localization algorithms is often dependent crucially upon how detailed the information obtained from reference nodes is.

The node localization algorithms we discussed include centroids, the use of overlapping geometric constraints, triangulation using distance estimates obtained using received signal strength and time difference of arrival, as well as AoA and pattern-matching approaches. For triangulation, TDoA techniques provide very accurate ranging at the expense of slightly more complex hardware. For RSS-based systems, an alternative to ranging-based triangulation for dense deployments is the ecolocation technique, which uses sequence-based decoding instead of absolute RSS values.

Network localization techniques include joint estimation techniques, iterative and collaborative multilateration, force-calculation, and multi-dimensional scaling. Even when no reference points are available, it is possible to construct a useful map of relative locations.

On the theoretical front, the Cramér–Rao bound on the error variance of unbiased estimators is useful in analyzing the performance of localization techniques. Rigidity theory is useful in formalizing the necessary and sufficient conditions for the existence of unique network localization.

Exercises

3.1 *Centroid:* Consider a node with unknown coordinates located in a square region of side d, with four reference nodes on the corners of the square. Consider three cases, when the reference beacons can be received within circles of radius: (a) $R = \frac{d}{\sqrt{2}}$, (b) $R = d$, and (c) $R = \sqrt{2}d$. In each case, identify the different unique centroid solutions that can be obtained (depending on the location of the unknown node) and the corresponding distinct regions. Estimate the worst case and average location estimate error in each case.

3.2 *Centroid versus proximity:* Consider three polygonal regions with reference nodes located at the vertices: (a) an equilateral triangle, (b) a square, and (c) a regular pentagon. Assume that an unknown node is uniformly likely to be located anywhere within the region, and that it is always within range of all reference nodes (and can correctly determine its nearest reference node if needed for a proximity determination). For each case compare the centroid localization technique with proximity localization,

4

Time synchronization

4.1 Overview

Given the need to coordinate the communication, computation, sensing, and actuation of distributed nodes, and the spatio-temporal nature of the monitored phenomena, it is no surprise that an accurate and consistent sense of time is essential in sensor networks. In this chapter, we shall discuss the many motivations for time synchronization, the challenges involved, as well as some of the solutions that have been proposed.

Distributed wireless sensor networks need time synchronization for a number of good reasons, some of which are described below:

1. **For time-stamping measurements:** Even the simplest data collection applications of sensor networks often require that sensor readings from different sensor nodes be provided with time stamps in addition to location information. This is particularly true whenever there may be a significant and unpredictable delay between when the measurement is taken at each source and when it is delivered to the sink/base station.
2. **For in-network signal processing:** Time stamps are needed to determine which information from different sources can be fused/aggregated within the network. Many collaborative signal processing algorithms, such as those for tracking unknown phenomena or targets, are coherent and require consistent and accurate synchronization.
3. **For localization:** Time-of-flight and TDoA-based ranging techniques used in node localization require good time synchronization.
4. **For cooperative communication:** Some physical layer multi-node cooperative communication techniques involve multiple transmitters transmitting

in-phase signals to a given receiver. Such techniques [105] have the potential to provide significant energy savings and robustness, but require tight synchronization.

5. **For medium-access:** TDMA-based medium-access schemes also require that nodes be synchronized so that they can be assigned distinct slots for collision-free communication.
6. **For sleep scheduling:** As we shall see in the following chapters, one of the most significant sources of energy savings is turning the radios of sensor devices off when they are not active. However, synchronization is needed to coordinate the sleep schedules of neighboring devices, so that they can communicate with each other efficiently.
7. **For coordinated actuation:** Advanced applications in which the network includes distributed actuators in addition to sensing require synchronization in order to coordinate the actuators through distributed control algorithms.

4.2 Key issues

The clock at each node consists of timer circuitry, often based on quartz crystal oscillators. The clock is incremented after each K ticks/interrupts of the timer. Practical timer circuits, particularly in low-end devices, are unstable and error prone. A model for clock non-ideality can be derived using the following expressions [89]. Let f_0 be the ideal frequency, Δf the frequency offset, d_f the drift in the frequency, and $r_f(t)$ an additional random error process, then the instantaneous oscillator frequency $f_i(t)$ of oscillator i at time t can be modeled as follows:

$$f_i(t) = f_0 + \Delta f + d_f t + r_f(t) \tag{4.1}$$

Then, assuming $t = 0$ as the initial reference time, the associated clock reads time $C_i(t)$ at time t, which is given as:

$$\begin{aligned} C_i(t) &= C_i(0) + \frac{1}{f_0}\int_0^t f_i(\tau)d\tau \\ &= C_i(0) + t + \frac{\Delta f}{f_0}t + \frac{d_f t^2}{2} + r_c(t) \end{aligned} \tag{4.2}$$

where $r_c(t)$ is the random clock error term corresponding to the error term $r_f(t)$ in the expression for oscillator frequency. Frequency drift and the random error term may be neglected to derive a simpler linear model for clock non-ideality:

$$C_i(t) = \alpha_i + \beta_i t \tag{4.3}$$

where α_i is the clock offset at the reference time $t = 0$ and β_i the clock drift (rate of change with respect to the ideal clock). The more stable and accurate the clock, the closer α_i is to 0, and the closer β_i is to 1. A clock is said to be *fast* if β_i is greater than 1, and *slow* otherwise.

Manufactured clocks are often specified with a maximum drift rate parameter ρ, such that $1 - \rho \leq \beta_i \leq 1 + \rho$. Motes, typical sensor nodes, have ρ values on the order of 40 ppm (parts per million), which corresponds to a drift rate of $\pm 40\mu$s per second.

Note that any two clocks that are synchronized once may drift from each other at a rate at most 2ρ. Hence, to keep their relative offset bounded by δ seconds at all times, the interval τ_{sync} corresponding to successive synchronization events between these clocks must be kept bounded: $\tau_{\text{sync}} \leq \delta/2\rho$.

Perhaps the simplest approach to time synchronization in a distributed system is through periodic broadcasts of a consistent global clock. In the US, the National Institute for Standards and Technology runs the radio stations WWV, WWVH, WWVB, that continuously broadcast timing signals based on atomic clocks. For instance WWVB, located at Fort Collins, Colorado, broadcasts timing signals on a 60 kHz carrier wave on a high power (50 kW) signal. Although the transmitter has an accuracy of about 1 μs, due to communication delays, synchronization around only 10μs is possible at receivers with this approach. While this can be implemented relatively inexpensively, the accuracy may not be sufficient for all purposes. Satellite-based GPS receivers can provide much better accuracy, of the order of 1μs or less, albeit at a higher expense, and they operate only in unobstructed environments. In some deployments it may be possible to use beacons from a subset of GPS-equipped nodes to provide synchronization to all nodes. In yet other networks, there may be no external sources of synchronization.

The requirements for time synchronization can vary greatly from application to application. In some cases the requirements may be very stringent – say 1μs synchronization between clocks – in others, it may be very lax – of the order of several milliseconds or even more. In some applications it will be necessary to keep all nodes synchronized globally to an external reference, while in others it will be sufficient to keep nodes synchronized locally and pair-wise to their immediate neighbors. In some applications it may be necessary to keep nodes synchronized at all times, and in other cases it may suffice only to know, *post facto*, the times when particular events occurred. Additional factors that determine the suitability of a particular synchronization approach to a given sensor network context include the corresponding energy costs, convergence times, and equipment costs.

4.3 Traditional approaches

Time synchronization is a long-studied subject in distributed systems, and a number of well-known algorithms have been developed for different conditions.[1]

For example, there is a well-known algorithm by Lamport [112] that provides a consistent ordering of all events in a distributed system, labelling each event x with a distinct time stamp L_x, such that: (a) $L_x \neq L_y$ for all unique events x and y, (b) if event x precedes event y within a node $L_x < L_y$, and (c) if x is the transmission of a message and y its reception at another node, $L_x < L_y$. These Lamport time stamps do not provide true causality. Say the true time of event x is indicated as T_x; then, while it is true that $T_x < T_y \rightarrow L_x < L_y$, it is not true that $L_x < L_y \rightarrow T_x < T_y$. Such a true causal ordering requires other approaches, such as the use of vector time stamps [209].

A fundamental technique for two-node clock synchronization is known as Cristian's algorithm [36]. A node A sends a request to node B (which has the reference clock) and receives back the value of B's clock, T_B. Node A records locally both the transmission time T_1 and the reception time T_2. This is illustrated in Figure 4.1.

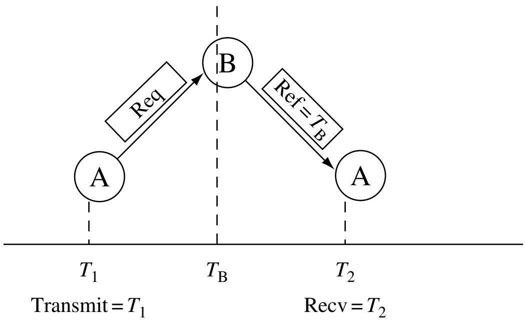


Figure 4.1 Cristian's synchronization algorithm

[1] The text by Tanenbaum and van Steen [209] provides a good discussion of many of these algorithms; we shall summarize these only briefly here.

In Cristian's time-synchronization algorithm, there are many sources of uncertainty and delay, which impact its accuracy. In general, message latency can be decomposed into four components, each of which contributes to uncertainty [108]:

- Send time – which includes any processing time and time taken to assemble and move the message to the link layer.
- Access time – which includes random delays while the message is buffered at the link layer due to contention and collisions.
- Propagation time – which is the time taken for point-to-point message travel. While negligible for a single link, this may be a dominant term over multiple hops if there is network congestion.
- Receive time – which is the time taken to process the message and record its arrival.

Good estimates of the message latency as well as the processing latency within the reference node must be obtained for Cristian's algorithm. The simplest estimate is to approximate the message propagation time as $(T_2 - T_1)/2$. If the processing delay is known to be I, then a better estimate is $(T_2 - T_1 - I)/2$. More sophisticated approaches take several round-trip delay samples and use minimum or mean delays after outlier removal.

The network time protocol (NTP) [138] is used widely on the Internet for time synchronization. It uses a hierarchy of reference time servers providing synchronization to querying clients, essentially using Cristian's algorithm.

4.4 Fine-grained clock synchronization

Several algorithms have been proposed for time synchronization in WSN. These all utilize time measurement-based message exchanges between nodes from time to time in order to synchronize the clocks on different nodes.

4.4.1 Reference broadcast synchronization (RBS)

The reference broadcast synchronization algorithm (RBS) [43] exploits the broadcast nature of wireless channels. RBS works as follows. Consider the scenario shown in Figure 4.2, with three nodes A, B, and C within the same broadcast domain. If B is a beacon node, it broadcasts the reference signal (which contains no timing information) that is received by both A and C simultaneously (neglecting propagation delay). The two receivers record the local time when the reference signal was received. Nodes A and C then exchange this local time stamp through separate messages. This is sufficient for the two receivers

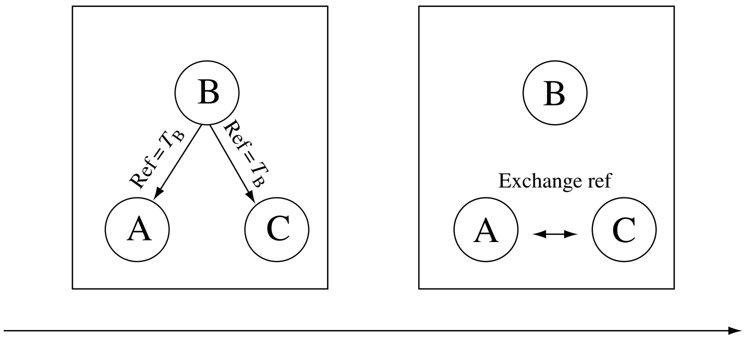


Figure 4.2 The reference broadcast synchronization (RBS) technique

to determine their relative offsets at the time of reference message reception. This basic scheme, which is quite similar to the CesiumSpray mechanism for synchronizing GPS-equipped nodes [213], can be extended to greater numbers of receivers. Improvements can be made by incorporating multiple reference broadcasts, which can help mitigate reception errors, as well as by estimating clock drifts.

A key feature of RBS is that it eliminates sender-side uncertainty completely. In scenarios where sender delays could be significant (particularly when time stamping has to be performed at the application layer instead of the link layer) this results in improved synchronization.

RBS can be extended to a multi-hop scenario as follows. If there are several separate reference beacons, each has its own broadcast domain that may overlap with the others. Receivers that lie in the overlapping region (in the broadcast domains of multiple references) provide "bridges" that allow nodes across these domains to determine the relationship between their local clocks, e.g., if nodes A and C are in range of reference B and nodes C and D are in range of reference E, then node C provides this bridge. In a large network, different paths through the temporal graph, representing connections between nodes sharing the same reference broadcast domain, provide different ways to convert times between arbitrary nodes. For efficiency, instead of computing these conversions *a priori* using global network information, these conversions can also be performed locally, on-the-fly, as packets traverse the network.

An interesting extension to the RBS technique is proposed and examined analytically in [103]. In particular, it is noted that the basic RBS scheme is composed only of a series of independent pair-wise synchronizations. This does not ensure global consistency in the following sense. Consider three nodes A, B, and C in the same domain, whose pair-wise offsets are determined through RBS.

There is no guarantee that the estimates obtained of $C_A(t) - C_B(t)$ and $C_B(t) - C_C(t)$ add up to the estimate obtained for $C_A(t) - C_C(t)$. An alternative technique has been developed for obtaining globally consistent minimum-variance pair-wise synchronization estimates, based on flow techniques for resistive networks [103].

4.4.2 Pair-wise sender–receiver synchronization (TPSN)

The timing-sync protocol for sensor networks (TPSN) [55] provides for classical sender–receiver synchronization, similar to Cristian's algorithm. As shown in Figure 4.3, node A transmits a message that is stamped locally at node A as T_1. This is received at node B, which stamps the reception time as its local time T_2. Node B then sends the packet back to node A, marking the transmission time locally at B as T_3. This is finally received at node A, which marks the reception time as T_4.

Let the clock offset between nodes A and B be Δ and the propagation delay between them be d. Then

$$T_2 = T_1 + \Delta + d \tag{4.4}$$

$$T_4 = T_3 - \Delta + d \tag{4.5}$$

which then result in the following:

$$\Delta = \frac{(T_2 - T_4) - (T_1 - T_3)}{2} \tag{4.6}$$

$$d = \frac{(T_2 + T_4) - (T_1 + T_3)}{2} \tag{4.7}$$

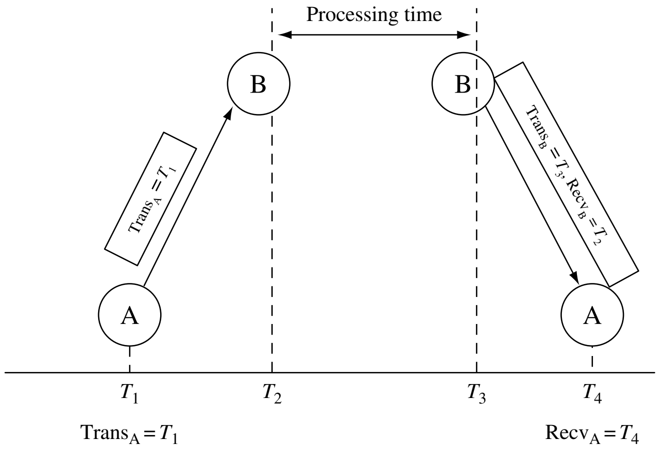


Figure 4.3 The basic sender–receiver synchronization technique used in TPSN

Network-wide time synchronization in TPSN is obtained level-by-level on a tree structure. Nodes at level 1 first synchronize with the root node. Nodes at level 2 then each synchronize with one node at level 1 and so on until all nodes in the network are synchronized with respect to the root.

Under the assumption that sender-side uncertainty can be mitigated by performing time stamping close to transmissions and receptions at the link layer, it is shown in [55] that the synchronization error with the pair-wise sender–receiver technique can actually provide twice the accuracy of the receiver–receiver technique used in RBS over a single link. This can potentially translate to even more significant gains over multiple hops.

The lightweight time synchronization (LTS) technique [68] is also a similar tree-based pair-wise sender–receiver synchronization technique.

4.4.3 Linear parameter-based synchronization

Another interesting approach to sender–receiver synchronization is the use of a linear model for clock behavior and corresponding parameter constraints to obtain synchronization [198]. This approach aims to provide deterministic bounds on relative clock drift and offset. Recall the simplified linear relationship for the behavior of a clock in equation (4.3). By combining that equation for nodes A and B together, we can derive the relationship between any two clocks A and B:

$$C_A(t) = \alpha_A - \alpha_B \frac{\beta_A}{\beta_B} + \frac{\beta_A}{\beta_B} C_B(t) \tag{4.8}$$

which can be re-written simply as

$$C_A(t) = \alpha_{AB} + \beta_{AB} C_B(t) \tag{4.9}$$

Assuming the same pair-wise message exchange as in TPSN for nodes A and B, we have that the transmission time T_1 and reception time T_4 are measured in node A's local clock, while reception time T_2 and transmission time T_3 are measured in node B's local clock. We therefore get the following temporal relationships:

$$T_1 < \alpha_{AB} + \beta_{AB} T_2 \tag{4.10}$$

$$T_4 > \alpha_{AB} + \beta_{AB} T_3 \tag{4.11}$$

The principle behind this approach to synchronization is to use these inequalities to determine constraints on the clock offset and drift. Each time such a pair-wise message takes place the expressions (4.10) and (4.11) provide additional constraints that together result in upper and lower bounds on the feasible

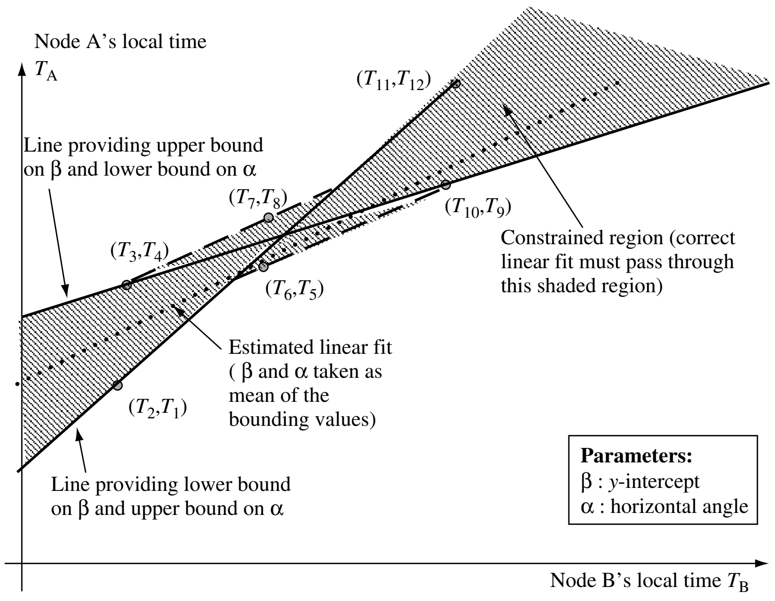

Node A's local time T_A
(T_{11},T_{12})
Line providing upper bound on β and lower bound on α
(T_7,T_8)
(T_{10},T_9)
(T_3,T_4)
Constrained region (correct linear fit must pass through this shaded region)
(T_6,T_5)
Estimated linear fit (β and α taken as mean of the bounding values)
(T_2,T_1)
Parameters:
β : y-intercept
α : horizontal angle
Line providing lower bound on β and upper bound on α
Node B's local time T_B

Figure 4.4 Visualization of bounding linear inequalities in the linear parameter-based technique

values of α_{AB} and β_{AB}. These bounds can be used to generate estimates of these quantities, which are needed for synchronization. The technique is illustrated in Figure 4.4.

To keep the storage and computational requirements of this approach bounded, there are two approaches. The first, called Tiny-Sync, keeps only four constraints at any time but can be suboptimal because it can discard potentially useful constraints. The second approach, called Mini-Sync, is guaranteed to provide optimal results and in theory may require a large number of constraints, but in practice is found to store no more than 40 constraints.

4.4.4 Flooding time synchronization protocol (FTSP)

The flooding time synchronization protocol (FTSP) [134] aims to further reduce the following sources of uncertainties, which exist in both RBS and TPSN:

1. **Interrupt handling time:** This is the delay in waiting for the processor to complete its current instruction before transferring the message in parts to the radio.

2. **Modulation/encoding time:** This is the time taken by the radio to perform modulation and encoding at the transmitter, and the corresponding demodulation and decoding at the receiver.

FTSP uses a broadcast from a single sender to synchronize multiple receivers. However, unlike RBS, the sender actually broadcasts a time measurement, and the receivers do not exchange messages among themselves. Each broadcast provides a synchronization point (a global–local time pair) to each receiver. FTSP has two main components:

1. Multiple time measurements: The sender takes several time stamp measurements during transmission, one at each byte boundary after a set of SYNC bytes used for byte alignment. These measurements are normalized by subtracting an appropriate multiple of the byte transmission time, and only the minimum of these multiple measurements is embedded into the message. At the receiver too, multiple time measurements are taken and the minimum of those is used as the receiver time. This serves to reduce the jitter significantly in interrupt handling and the (de)coding and (de)modulation times. With as few as six time stamps, an order of magnitude improvement in precision can be obtained on a Mica Mote platform (from the order of tens of microseconds to the order of about one microsecond).
2. Flooded messaging: To propagate the synchronization information, a flooding approach is used. First, a single uniquely identifiable node in the network provides the global clock. The reception of each broadcast message allows a receiver to accumulate a reference synchronization point. When a receiver accumulates several reference points, it becomes synchronized itself (e.g. using a regression line to estimate the local clock drift). Nodes can collect reference points either from the global reference node, or from other nodes that are already synchronized. The frequency of the flooding provides a tradeoff between synchronization accuracy and overhead.

4.4.5 Predictive time synchronization

The simple linear model of equation (4.3) is only reasonable for very short time intervals. In the real world, clock drift can vary over time quite drastically due to environmental temperature and humidity changes. This is the reason clock drifts must be continually reassessed. The naïve approach to this reassessment is to re-synchronize nodes periodically at the same interval. However, a static synchronization period must be chosen conservatively to accommodate a range of environments. This does not take into account the possibility of temporal correlations in clock drift. It will thus incur an unnecessarily high overhead in many cases.

This problem is addressed by the predictive synchronization mechanism [56] in which the frequency of inter-node time sampling is adaptively adjusted. It has been determined through an empirical study that environments are characterized by a time constant T over which drift rates are highly correlated, which can be determined through a learning phase. Depending on the time sampling period S, a window of T/S prior sample measurements is used in this technique not only to predict the clock drift (through linear regression), but also to estimate the error in the prediction. A MIMD technique is used to adapt the sampling period: if the prediction error is above a desirable threshold, the sampling period S is reduced multiplicatively; and if it is below threshold, the sampling period is increased accordingly. This adaptive scheme provides for robust long-term synchronization in a self-configuring manner.

4.5 Coarse-grained data synchronization

The Wisden system (a wireless sensor network system for structural-response data acquisition [231]) presents an excellent lightweight alternative to clock-synchronization approaches that is suitable for data-gathering applications. The approach is to collect and record latency measurements within each packet in a special residence time field as the packet propagates through the network. The estimate of delivery latency recorded in the packet is then used to provide a retroactive time stamp eventually at the base station when the packet arrives.

In this approach, only the base station is required to have an accurate reference clock. Since the radio propagation delays are insignificant, what is measured at each hop is actually the time that the packet spends at each node – which can be of the order of milliseconds due to queuing and processing delays. Say the time spent by packet i at the kth intermediate node on an $n+1$ hop-path to the destination is λ_k^i, let the time at the base station d when the packet is received be T_d^i, then the packet's original contents are time stamped to have been generated at source s at

$$T_s^i = T_d^i - \sum_{k=1}^{n} \lambda_k^i \tag{4.12}$$

This approach assumes that time stamps can be added as close to the packet transmission and reception as possible at the link layer. It is thus robust to many of the sources of latency uncertainty that contribute to error in other synchronization approaches. However, it is vulnerable to varying clock drifts at the intermediate nodes, particularly when packet residence times within the network are long. For example, it has been estimated that with a 10 ppm drift, data samples cannot reside for more than 15 minutes before accumulating 10 ms of error in the time stamp.

4.6 Summary

Like localization, time synchronization is also a core configuration problem in WSNs. It is a fundamental service building block useful for many network functions, including time stamping of sensor measurements, coherent distributed signal processing, cooperative communication, medium-access, and sleep scheduling. Synchronization is necessitated by the random clock drifts that vary depending on hardware and environmental conditions.

Two approaches to fine-grained time synchronization are the receiver–receiver synchronization technique of RBS, and the more traditional sender–receiver approach of TPSN. While the latter provides for greater accuracy on a single link, RBS has the advantage that multiple receivers can be synchronized with fewer messages. It has been shown that these can provide synchronization of the order of tens of micro-seconds. The flooding-time synchronization protocol further improves performance by another order of magnitude by reducing uncertainties due to jitter in interrupt handling and coding/modulation. Thus it appears that even fairly demanding synchronization requirements can be met in principle through such algorithms. However, there is an energy–accuracy tradeoff involved in long-term synchronization, because the accuracy is determined by the frequency with which nodes are periodically re-synchronized. It has been shown that adaptive prediction-based drift-estimation techniques can reduce this overhead further.

For some applications, instead of using inter-node synchronization, coarse-grained data time stamps can be obtained by timing packets as they move through the network and by performing a simple calculation at the final destination.

Exercises

4.1 *Unsynchronized clock drift:* Plot the variation of a clock's measured time with respect to real time, assuming $f_0 = 1$, $\Delta f = 0.1$, $d_f = -0.01$.

4.2 *Synchronization frequency:* How frequently should a clock with $\Delta f = 0.3$, $d_f = 0$ be synchronized with an ideal clock to ensure that it does not stray more than one time unit from the real clock?

4.3 *Communication scaling:* Assume there are n nodes within range of each other, among which one has an ideal clock. How many messages in total are required to synchronize the clocks of all nodes in case of (a) RBS and (b) TPSN? What does this suggest about the scalability of these techniques?

4.4 *Error propagation:* Consider n nodes in a chain, numbered 1 to n left to right. Assume that, starting with node 2, each node's clock is successively synchronized pair-wise with the node on its left. Due to processing-time uncertainties, assume that the clock difference between each pair of neighboring nodes after synchronization (i.e. the synchronization error) is not deterministically zero, but rather a zero-mean Gaussian with variance σ^2. What is the form of the end-to-end synchronization error across n hops?

4.5 *Linear parameter-based synchronization:* Consider the following three sets of four time stamps on messages exchanged between the nodes A and B: [(1,2,3,8), (9,10,11,12), (13,18,19,20)]. As in Figure 4.4, draw a diagram showing the corresponding linear inequalities, indicating the constrained region as well as the lines corresponding to the bounds on α_{AB} and β_{AB}. What are these bounds and the corresponding mean estimate for synchronization?

4.6 *Data-stamping error accumulation:* Assuming a worst case drift of 40 ppm at each intermediate node, how long can a packet stay within the network when using the Wisden data-stamping technique before the total error exceeds 1 second?

4

Time synchronization

4.1 Overview

Given the need to coordinate the communication, computation, sensing, and actuation of distributed nodes, and the spatio-temporal nature of the monitored phenomena, it is no surprise that an accurate and consistent sense of time is essential in sensor networks. In this chapter, we shall discuss the many motivations for time synchronization, the challenges involved, as well as some of the solutions that have been proposed.

Distributed wireless sensor networks need time synchronization for a number of good reasons, some of which are described below:

1. **For time-stamping measurements:** Even the simplest data collection applications of sensor networks often require that sensor readings from different sensor nodes be provided with time stamps in addition to location information. This is particularly true whenever there may be a significant and unpredictable delay between when the measurement is taken at each source and when it is delivered to the sink/base station.
2. **For in-network signal processing:** Time stamps are needed to determine which information from different sources can be fused/aggregated within the network. Many collaborative signal processing algorithms, such as those for tracking unknown phenomena or targets, are coherent and require consistent and accurate synchronization.
3. **For localization:** Time-of-flight and TDoA-based ranging techniques used in node localization require good time synchronization.
4. **For cooperative communication:** Some physical layer multi-node cooperative communication techniques involve multiple transmitters transmitting

in-phase signals to a given receiver. Such techniques [105] have the potential to provide significant energy savings and robustness, but require tight synchronization.

5. **For medium-access:** TDMA-based medium-access schemes also require that nodes be synchronized so that they can be assigned distinct slots for collision-free communication.
6. **For sleep scheduling:** As we shall see in the following chapters, one of the most significant sources of energy savings is turning the radios of sensor devices off when they are not active. However, synchronization is needed to coordinate the sleep schedules of neighboring devices, so that they can communicate with each other efficiently.
7. **For coordinated actuation:** Advanced applications in which the network includes distributed actuators in addition to sensing require synchronization in order to coordinate the actuators through distributed control algorithms.

4.2 Key issues

The clock at each node consists of timer circuitry, often based on quartz crystal oscillators. The clock is incremented after each K ticks/interrupts of the timer. Practical timer circuits, particularly in low-end devices, are unstable and error prone. A model for clock non-ideality can be derived using the following expressions [89]. Let f_0 be the ideal frequency, Δf the frequency offset, d_f the drift in the frequency, and $r_f(t)$ an additional random error process, then the instantaneous oscillator frequency $f_i(t)$ of oscillator i at time t can be modeled as follows:

$$f_i(t) = f_0 + \Delta f + d_f t + r_f(t) \tag{4.1}$$

Then, assuming $t = 0$ as the initial reference time, the associated clock reads time $C_i(t)$ at time t, which is given as:

$$\begin{aligned} C_i(t) &= C_i(0) + \frac{1}{f_0}\int_0^t f_i(\tau)d\tau \\ &= C_i(0) + t + \frac{\Delta f}{f_0}t + \frac{d_f t^2}{2} + r_c(t) \end{aligned} \tag{4.2}$$

where $r_c(t)$ is the random clock error term corresponding to the error term $r_f(t)$ in the expression for oscillator frequency. Frequency drift and the random error term may be neglected to derive a simpler linear model for clock non-ideality:

$$C_i(t) = \alpha_i + \beta_i t \tag{4.3}$$

where α_i is the clock offset at the reference time $t = 0$ and β_i the clock drift (rate of change with respect to the ideal clock). The more stable and accurate the clock, the closer α_i is to 0, and the closer β_i is to 1. A clock is said to be *fast* if β_i is greater than 1, and *slow* otherwise.

Manufactured clocks are often specified with a maximum drift rate parameter ρ, such that $1 - \rho \leq \beta_i \leq 1 + \rho$. Motes, typical sensor nodes, have ρ values on the order of 40 ppm (parts per million), which corresponds to a drift rate of $\pm 40\mu$s per second.

Note that any two clocks that are synchronized once may drift from each other at a rate at most 2ρ. Hence, to keep their relative offset bounded by δ seconds at all times, the interval τ_{sync} corresponding to successive synchronization events between these clocks must be kept bounded: $\tau_{\text{sync}} \leq \delta/2\rho$.

Perhaps the simplest approach to time synchronization in a distributed system is through periodic broadcasts of a consistent global clock. In the US, the National Institute for Standards and Technology runs the radio stations WWV, WWVH, WWVB, that continuously broadcast timing signals based on atomic clocks. For instance WWVB, located at Fort Collins, Colorado, broadcasts timing signals on a 60 kHz carrier wave on a high power (50 kW) signal. Although the transmitter has an accuracy of about $1\,\mu$s, due to communication delays, synchronization around only 10μs is possible at receivers with this approach. While this can be implemented relatively inexpensively, the accuracy may not be sufficient for all purposes. Satellite-based GPS receivers can provide much better accuracy, of the order of 1μs or less, albeit at a higher expense, and they operate only in unobstructed environments. In some deployments it may be possible to use beacons from a subset of GPS-equipped nodes to provide synchronization to all nodes. In yet other networks, there may be no external sources of synchronization.

The requirements for time synchronization can vary greatly from application to application. In some cases the requirements may be very stringent – say 1μs synchronization between clocks – in others, it may be very lax – of the order of several milliseconds or even more. In some applications it will be necessary to keep all nodes synchronized globally to an external reference, while in others it will be sufficient to keep nodes synchronized locally and pair-wise to their immediate neighbors. In some applications it may be necessary to keep nodes synchronized at all times, and in other cases it may suffice only to know, *post facto*, the times when particular events occurred. Additional factors that determine the suitability of a particular synchronization approach to a given sensor network context include the corresponding energy costs, convergence times, and equipment costs.

4.3 Traditional approaches

Time synchronization is a long-studied subject in distributed systems, and a number of well-known algorithms have been developed for different conditions.[1]

For example, there is a well-known algorithm by Lamport [112] that provides a consistent ordering of all events in a distributed system, labelling each event x with a distinct time stamp L_x, such that: (a) $L_x \neq L_y$ for all unique events x and y, (b) if event x precedes event y within a node $L_x < L_y$, and (c) if x is the transmission of a message and y its reception at another node, $L_x < L_y$. These Lamport time stamps do not provide true causality. Say the true time of event x is indicated as T_x; then, while it is true that $T_x < T_y \rightarrow L_x < L_y$, it is not true that $L_x < L_y \rightarrow T_x < T_y$. Such a true causal ordering requires other approaches, such as the use of vector time stamps [209].

A fundamental technique for two-node clock synchronization is known as Cristian's algorithm [36]. A node A sends a request to node B (which has the reference clock) and receives back the value of B's clock, T_B. Node A records locally both the transmission time T_1 and the reception time T_2. This is illustrated in Figure 4.1.

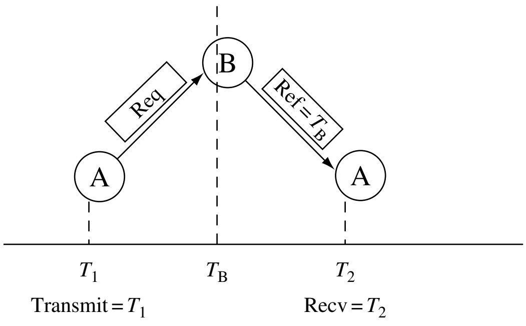


Figure 4.1 Cristian's synchronization algorithm

[1] The text by Tanenbaum and van Steen [209] provides a good discussion of many of these algorithms; we shall summarize these only briefly here.

In Cristian's time-synchronization algorithm, there are many sources of uncertainty and delay, which impact its accuracy. In general, message latency can be decomposed into four components, each of which contributes to uncertainty [108]:

- Send time – which includes any processing time and time taken to assemble and move the message to the link layer.
- Access time – which includes random delays while the message is buffered at the link layer due to contention and collisions.
- Propagation time – which is the time taken for point-to-point message travel. While negligible for a single link, this may be a dominant term over multiple hops if there is network congestion.
- Receive time – which is the time taken to process the message and record its arrival.

Good estimates of the message latency as well as the processing latency within the reference node must be obtained for Cristian's algorithm. The simplest estimate is to approximate the message propagation time as $(T_2 - T_1)/2$. If the processing delay is known to be I, then a better estimate is $(T_2 - T_1 - I)/2$. More sophisticated approaches take several round-trip delay samples and use minimum or mean delays after outlier removal.

The network time protocol (NTP) [138] is used widely on the Internet for time synchronization. It uses a hierarchy of reference time servers providing synchronization to querying clients, essentially using Cristian's algorithm.

4.4 Fine-grained clock synchronization

Several algorithms have been proposed for time synchronization in WSN. These all utilize time measurement-based message exchanges between nodes from time to time in order to synchronize the clocks on different nodes.

4.4.1 Reference broadcast synchronization (RBS)

The reference broadcast synchronization algorithm (RBS) [43] exploits the broadcast nature of wireless channels. RBS works as follows. Consider the scenario shown in Figure 4.2, with three nodes A, B, and C within the same broadcast domain. If B is a beacon node, it broadcasts the reference signal (which contains no timing information) that is received by both A and C simultaneously (neglecting propagation delay). The two receivers record the local time when the reference signal was received. Nodes A and C then exchange this local time stamp through separate messages. This is sufficient for the two receivers

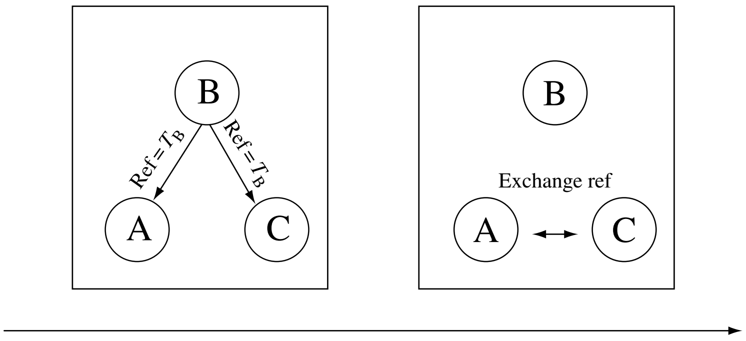


Figure 4.2 The reference broadcast synchronization (RBS) technique

to determine their relative offsets at the time of reference message reception. This basic scheme, which is quite similar to the CesiumSpray mechanism for synchronizing GPS-equipped nodes [213], can be extended to greater numbers of receivers. Improvements can be made by incorporating multiple reference broadcasts, which can help mitigate reception errors, as well as by estimating clock drifts.

A key feature of RBS is that it eliminates sender-side uncertainty completely. In scenarios where sender delays could be significant (particularly when time stamping has to be performed at the application layer instead of the link layer) this results in improved synchronization.

RBS can be extended to a multi-hop scenario as follows. If there are several separate reference beacons, each has its own broadcast domain that may overlap with the others. Receivers that lie in the overlapping region (in the broadcast domains of multiple references) provide "bridges" that allow nodes across these domains to determine the relationship between their local clocks, e.g., if nodes A and C are in range of reference B and nodes C and D are in range of reference E, then node C provides this bridge. In a large network, different paths through the temporal graph, representing connections between nodes sharing the same reference broadcast domain, provide different ways to convert times between arbitrary nodes. For efficiency, instead of computing these conversions *a priori* using global network information, these conversions can also be performed locally, on-the-fly, as packets traverse the network.

An interesting extension to the RBS technique is proposed and examined analytically in [103]. In particular, it is noted that the basic RBS scheme is composed only of a series of independent pair-wise synchronizations. This does not ensure global consistency in the following sense. Consider three nodes A, B, and C in the same domain, whose pair-wise offsets are determined through RBS.

There is no guarantee that the estimates obtained of $C_A(t) - C_B(t)$ and $C_B(t) - C_C(t)$ add up to the estimate obtained for $C_A(t) - C_C(t)$. An alternative technique has been developed for obtaining globally consistent minimum-variance pair-wise synchronization estimates, based on flow techniques for resistive networks [103].

4.4.2 Pair-wise sender–receiver synchronization (TPSN)

The timing-sync protocol for sensor networks (TPSN) [55] provides for classical sender–receiver synchronization, similar to Cristian's algorithm. As shown in Figure 4.3, node A transmits a message that is stamped locally at node A as T_1. This is received at node B, which stamps the reception time as its local time T_2. Node B then sends the packet back to node A, marking the transmission time locally at B as T_3. This is finally received at node A, which marks the reception time as T_4.

Let the clock offset between nodes A and B be Δ and the propagation delay between them be d. Then

$$T_2 = T_1 + \Delta + d \tag{4.4}$$

$$T_4 = T_3 - \Delta + d \tag{4.5}$$

which then result in the following:

$$\Delta = \frac{(T_2 - T_4) - (T_1 - T_3)}{2} \tag{4.6}$$

$$d = \frac{(T_2 + T_4) - (T_1 + T_3)}{2} \tag{4.7}$$

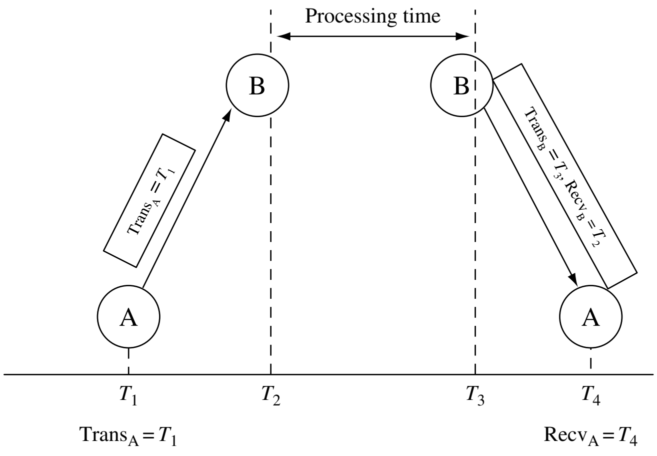


Figure 4.3 The basic sender–receiver synchronization technique used in TPSN

Network-wide time synchronization in TPSN is obtained level-by-level on a tree structure. Nodes at level 1 first synchronize with the root node. Nodes at level 2 then each synchronize with one node at level 1 and so on until all nodes in the network are synchronized with respect to the root.

Under the assumption that sender-side uncertainty can be mitigated by performing time stamping close to transmissions and receptions at the link layer, it is shown in [55] that the synchronization error with the pair-wise sender–receiver technique can actually provide twice the accuracy of the receiver–receiver technique used in RBS over a single link. This can potentially translate to even more significant gains over multiple hops.

The lightweight time synchronization (LTS) technique [68] is also a similar tree-based pair-wise sender–receiver synchronization technique.

4.4.3 Linear parameter-based synchronization

Another interesting approach to sender–receiver synchronization is the use of a linear model for clock behavior and corresponding parameter constraints to obtain synchronization [198]. This approach aims to provide deterministic bounds on relative clock drift and offset. Recall the simplified linear relationship for the behavior of a clock in equation (4.3). By combining that equation for nodes A and B together, we can derive the relationship between any two clocks A and B:

$$C_{\mathrm{A}}(t) = \alpha_{\mathrm{A}} - \alpha_{\mathrm{B}}\frac{\beta_{\mathrm{A}}}{\beta_{\mathrm{B}}} + \frac{\beta_{\mathrm{A}}}{\beta_{\mathrm{B}}}C_{\mathrm{B}}(t) \tag{4.8}$$

which can be re-written simply as

$$C_{\mathrm{A}}(t) = \alpha_{\mathrm{AB}} + \beta_{\mathrm{AB}}C_{\mathrm{B}}(t) \tag{4.9}$$

Assuming the same pair-wise message exchange as in TPSN for nodes A and B, we have that the transmission time T_1 and reception time T_4 are measured in node A's local clock, while reception time T_2 and transmission time T_3 are measured in node B's local clock. We therefore get the following temporal relationships:

$$T_1 < \alpha_{\mathrm{AB}} + \beta_{\mathrm{AB}}T_2 \tag{4.10}$$

$$T_4 > \alpha_{\mathrm{AB}} + \beta_{\mathrm{AB}}T_3 \tag{4.11}$$

The principle behind this approach to synchronization is to use these inequalities to determine constraints on the clock offset and drift. Each time such a pair-wise message takes place the expressions (4.10) and (4.11) provide additional constraints that together result in upper and lower bounds on the feasible

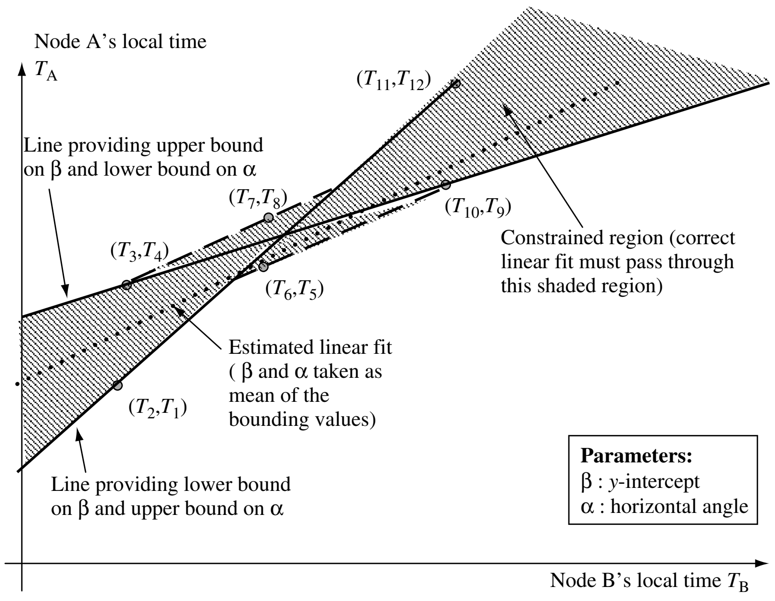

Node A's local time T_A
(T_{11},T_{12})
Line providing upper bound on β and lower bound on α
(T_7,T_8)
(T_{10},T_9)
Constrained region (correct linear fit must pass through this shaded region)
(T_3,T_4)
(T_6,T_5)
Estimated linear fit (β and α taken as mean of the bounding values)
(T_2,T_1)
Parameters:
β : y-intercept
α : horizontal angle
Line providing lower bound on β and upper bound on α
Node B's local time T_B

Figure 4.4 Visualization of bounding linear inequalities in the linear parameter-based technique

values of α_{AB} and β_{AB}. These bounds can be used to generate estimates of these quantities, which are needed for synchronization. The technique is illustrated in Figure 4.4.

To keep the storage and computational requirements of this approach bounded, there are two approaches. The first, called Tiny-Sync, keeps only four constraints at any time but can be suboptimal because it can discard potentially useful constraints. The second approach, called Mini-Sync, is guaranteed to provide optimal results and in theory may require a large number of constraints, but in practice is found to store no more than 40 constraints.

4.4.4 Flooding time synchronization protocol (FTSP)

The flooding time synchronization protocol (FTSP) [134] aims to further reduce the following sources of uncertainties, which exist in both RBS and TPSN:

1. **Interrupt handling time:** This is the delay in waiting for the processor to complete its current instruction before transferring the message in parts to the radio.

2. **Modulation/encoding time:** This is the time taken by the radio to perform modulation and encoding at the transmitter, and the corresponding demodulation and decoding at the receiver.

FTSP uses a broadcast from a single sender to synchronize multiple receivers. However, unlike RBS, the sender actually broadcasts a time measurement, and the receivers do not exchange messages among themselves. Each broadcast provides a synchronization point (a global–local time pair) to each receiver. FTSP has two main components:

1. Multiple time measurements: The sender takes several time stamp measurements during transmission, one at each byte boundary after a set of SYNC bytes used for byte alignment. These measurements are normalized by subtracting an appropriate multiple of the byte transmission time, and only the minimum of these multiple measurements is embedded into the message. At the receiver too, multiple time measurements are taken and the minimum of those is used as the receiver time. This serves to reduce the jitter significantly in interrupt handling and the (de)coding and (de)modulation times. With as few as six time stamps, an order of magnitude improvement in precision can be obtained on a Mica Mote platform (from the order of tens of microseconds to the order of about one microsecond).
2. Flooded messaging: To propagate the synchronization information, a flooding approach is used. First, a single uniquely identifiable node in the network provides the global clock. The reception of each broadcast message allows a receiver to accumulate a reference synchronization point. When a receiver accumulates several reference points, it becomes synchronized itself (e.g. using a regression line to estimate the local clock drift). Nodes can collect reference points either from the global reference node, or from other nodes that are already synchronized. The frequency of the flooding provides a tradeoff between synchronization accuracy and overhead.

4.4.5 Predictive time synchronization

The simple linear model of equation (4.3) is only reasonable for very short time intervals. In the real world, clock drift can vary over time quite drastically due to environmental temperature and humidity changes. This is the reason clock drifts must be continually reassessed. The naïve approach to this reassessment is to re-synchronize nodes periodically at the same interval. However, a static synchronization period must be chosen conservatively to accommodate a range of environments. This does not take into account the possibility of temporal correlations in clock drift. It will thus incur an unnecessarily high overhead in many cases.

This problem is addressed by the predictive synchronization mechanism [56] in which the frequency of inter-node time sampling is adaptively adjusted. It has been determined through an empirical study that environments are characterized by a time constant T over which drift rates are highly correlated, which can be determined through a learning phase. Depending on the time sampling period S, a window of T/S prior sample measurements is used in this technique not only to predict the clock drift (through linear regression), but also to estimate the error in the prediction. A MIMD technique is used to adapt the sampling period: if the prediction error is above a desirable threshold, the sampling period S is reduced multiplicatively; and if it is below threshold, the sampling period is increased accordingly. This adaptive scheme provides for robust long-term synchronization in a self-configuring manner.

4.5 Coarse-grained data synchronization

The Wisden system (a wireless sensor network system for structural-response data acquisition [231]) presents an excellent lightweight alternative to clock-synchronization approaches that is suitable for data-gathering applications. The approach is to collect and record latency measurements within each packet in a special residence time field as the packet propagates through the network. The estimate of delivery latency recorded in the packet is then used to provide a retroactive time stamp eventually at the base station when the packet arrives.

In this approach, only the base station is required to have an accurate reference clock. Since the radio propagation delays are insignificant, what is measured at each hop is actually the time that the packet spends at each node – which can be of the order of milliseconds due to queuing and processing delays. Say the time spent by packet i at the kth intermediate node on an $n+1$ hop-path to the destination is λ_k^i, let the time at the base station d when the packet is received be T_d^i, then the packet's original contents are time stamped to have been generated at source s at

$$T_s^i = T_d^i - \sum_{k=1}^{n} \lambda_k^i \tag{4.12}$$

This approach assumes that time stamps can be added as close to the packet transmission and reception as possible at the link layer. It is thus robust to many of the sources of latency uncertainty that contribute to error in other synchronization approaches. However, it is vulnerable to varying clock drifts at the intermediate nodes, particularly when packet residence times within the network are long. For example, it has been estimated that with a 10 ppm drift, data samples cannot reside for more than 15 minutes before accumulating 10 ms of error in the time stamp.

4.6 Summary

Like localization, time synchronization is also a core configuration problem in WSNs. It is a fundamental service building block useful for many network functions, including time stamping of sensor measurements, coherent distributed signal processing, cooperative communication, medium-access, and sleep scheduling. Synchronization is necessitated by the random clock drifts that vary depending on hardware and environmental conditions.

Two approaches to fine-grained time synchronization are the receiver–receiver synchronization technique of RBS, and the more traditional sender–receiver approach of TPSN. While the latter provides for greater accuracy on a single link, RBS has the advantage that multiple receivers can be synchronized with fewer messages. It has been shown that these can provide synchronization of the order of tens of micro-seconds. The flooding-time synchronization protocol further improves performance by another order of magnitude by reducing uncertainties due to jitter in interrupt handling and coding/modulation. Thus it appears that even fairly demanding synchronization requirements can be met in principle through such algorithms. However, there is an energy–accuracy tradeoff involved in long-term synchronization, because the accuracy is determined by the frequency with which nodes are periodically re-synchronized. It has been shown that adaptive prediction-based drift-estimation techniques can reduce this overhead further.

For some applications, instead of using inter-node synchronization, coarse-grained data time stamps can be obtained by timing packets as they move through the network and by performing a simple calculation at the final destination.

Exercises

4.1 *Unsynchronized clock drift:* Plot the variation of a clock's measured time with respect to real time, assuming $f_0 = 1$, $\Delta f = 0.1$, $d_f = -0.01$.

4.2 *Synchronization frequency:* How frequently should a clock with $\Delta f = 0.3$, $d_f = 0$ be synchronized with an ideal clock to ensure that it does not stray more than one time unit from the real clock?

4.3 *Communication scaling:* Assume there are n nodes within range of each other, among which one has an ideal clock. How many messages in total are required to synchronize the clocks of all nodes in case of (a) RBS and (b) TPSN? What does this suggest about the scalability of these techniques?

4.4 *Error propagation:* Consider n nodes in a chain, numbered 1 to n left to right. Assume that, starting with node 2, each node's clock is successively synchronized pair-wise with the node on its left. Due to processing-time uncertainties, assume that the clock difference between each pair of neighboring nodes after synchronization (i.e. the synchronization error) is not deterministically zero, but rather a zero-mean Gaussian with variance σ^2. What is the form of the end-to-end synchronization error across n hops?

4.5 *Linear parameter-based synchronization:* Consider the following three sets of four time stamps on messages exchanged between the nodes A and B: [(1,2,3,8), (9,10,11,12), (13,18,19,20)]. As in Figure 4.4, draw a diagram showing the corresponding linear inequalities, indicating the constrained region as well as the lines corresponding to the bounds on α_{AB} and β_{AB}. What are these bounds and the corresponding mean estimate for synchronization?

4.6 *Data-stamping error accumulation:* Assuming a worst case drift of 40 ppm at each intermediate node, how long can a packet stay within the network when using the Wisden data-stamping technique before the total error exceeds 1 second?

only wake-up just before their assigned GTS slots. The communication during CAP is a simple CSMA-CA algorithm, which allows for a small backoff period to reduce idle listening energy consumption. A performance evaluation of this protocol and its various settings and parameters can be found in [126].

While the IEEE 802.15.4 can, in theory, be used for other topologies, the beacon-enabled mode is not defined for them. In the rest of the chapter we will concern ourselves with both contention-based and schedule-based energy-efficient MAC protocols that are relevant to multi-hop wireless networks.

6.3 Energy efficiency in MAC protocols

Energy efficiency is obtained in MAC protocols essentially by turning off the radio to sleep mode whenever possible, to save on radio power consumption.

6.3.1 Power management in IEEE 802.11

There exist power management options in the infrastructure mode for 802.11. Nodes inform the access point (AP) when they wish to enter sleep mode so that any messages for them can be buffered at the AP. The nodes periodically wake-up to check for these buffered messages. Energy savings are thus provided at the expense of lower throughput and higher latency.

6.3.2 Power aware medium-access with signalling (PAMAS)

The PAMAS (power aware multi-access protocol with signalling) [199] is an extension of the MACA technique, where the RTS/CTS signalling is carried out on a separate radio channel from the data exchange. It is one of the first power aware MAC protocols proposed for multi-hop wireless networks. In PAMAS, nodes turn off the radio (go to sleep) whenever they can neither receive nor transmit successfully. Specifically they go to sleep whenever they overhear a neighbor transmitting to another node, or if they determine through the control channel RTS/CTS signaling that one of their neighbors is receiving. The duration of the sleep mode is set to the length of the ongoing transmissions indicated by the control signals received on the secondary channel. If a transmission is started while a node is in sleep mode, upon wake-up the node sends probe signals to determine the duration of the ongoing transmission and how long it can go back to sleep. In PAMAS, a node will only be put to sleep when it is inhibited from transmitting/receiving anyway, so that the delay/throughput performances of the network are not affected adversely. However, there can still be considerable energy wastage in the idle reception mode (i.e. the condition when a node has no packets to send and there is no activity on the channel).

6.3.3 Minimizing the idle reception energy costs

While PAMAS provides ways to save energy on overhearing, further energy savings are possible by reducing idle receptions. The key challenge is to allow receivers to sleep a majority of the time, while still ensuring that a node is awake and receiving when a packet intended for it is being transmitted. Based on the methods to solve this problem, there are essentially two classes of contention-based sensor network MAC protocols.

The first approach is completely asynchronous and relies solely on the use of an additional radio or periodic low-power listening techniques to ensure that the receiver is woken up for an incoming transmission intended for it. The second approach, with many variants, uses periodic duty-cycled sleep schedules for nodes. Most often the schedules are coordinated in such a way that transmitters know in advance when their intended receiver will be awake.

6.4 Asynchronous sleep techniques

In these techniques nodes normally keep their radios in sleep mode as a default, waking up briefly only to check for traffic or to send/receive messages.

6.4.1 Secondary wake-up radio

Nodes need to be able to sleep to save energy when they do not have any communication activity and be awake to participate in any necessary communications. The first solution is a hardware one – equipping each sensor node with two radios. In such a hardware design, the primary radio is the main data radio, which remains asleep by default. The secondary radio is a low-power wake-up radio that remains on at all times. Such an idea is described in the Pico Radio project [166], as well as by Shih *et al.* [195]. If the wake-up radio of a node receives a wake-up signal from another node, it responds by waking up the primary radio to begin receiving. This ensures that the primary radio is active only when the node has data to send or receive. The underlying assumption motivating such a design is that, since the wake-up radio need not do much sophisticated signal processing, it can be designed to be extremely low power. A tradeoff, however, is that all nodes in the broadcast domain of the transmitting node may be woken up.

8

Energy-efficient and robust routing

8.1 Overview

Information routing in wireless sensor networks can be made robust and energy-efficient by taking into account a number of pieces of state information available locally within the network.

1. **Link quality:** As we discussed in Chapter 5, link quality metrics (e.g. packet reception rates) obtained through periodic monitoring are very useful in making routing decisions.
2. **Link distance:** Particularly in case of highly dynamic rapidly fading environments, if link monitoring incurs too high an overhead, link distances can be useful indicators of link quality and energy consumption.
3. **Residual energy:** In order to extend network lifetimes it may be desirable to avoid routing through nodes with low residual energy.
4. **Location information:** If relative or absolute location information is available, geographic routing techniques may be used to minimize routing overhead.
5. **Mobility information:** Recorded information about the nearest static sensor node near a mobile node is also useful for routing.

We examine in this chapter several routing techniques that utilize such information to provide energy efficiency and robustness.

8.2 Metric-based approaches

Robustness can be provided by selecting routes that minimize end-to-end retransmissions or failure probabilities. This requires the selection of a suitable metric for each link.

8.2.1 The ETX metric

If all wireless links are considered to be ideal error-free links, then routing data through the network along shortest hop–count paths may be appropriate. However, the use of shortest hop–count paths would require the distances of the component links to be high. In a practical wireless system, these links are highly likely to be error-prone and lie in the transitional region. Therefore, the shortest hop–count path strategy will perform quite poorly in realistic settings. This has been verified in a study [39], which presents a metric suitable for robust routing in a wireless network. This metric, called ETX, minimizes the expected number of total transmissions on a path. Independently, an almost identical metric called the minimum transmission metric was also developed for WSN [225].

It is assumed that all transmissions are performed with ARQ in the form of simple ACK signals for each successfully delivered packet. Let d_f be the packet reception rate (probability of successful delivery) on a link in the forward direction, and d_r the probability that the corresponding ACK is received in the reverse direction. Then, assuming each packet transmission can be treated as a Bernoulli trial, the expected number of transmissions required for successful delivery of a packet on the link is:

$$ETX = \frac{1}{d_\mathrm{f} \cdot d_\mathrm{r}} \tag{8.1}$$

This metric for a single link can then be incorporated into any relevant routing protocol, so that end-to-end paths are constructed to minimize the sum of ETX on each link on the path, i.e. the total expected number of transmissions on the route.

Figure 8.1 shows three routes between a given source A and destination B, each with different numbers of hops with the labelled link qualities (only the forward probabilities are shown, assume the reverse probabilities are all $d_\mathrm{r} = 1$).

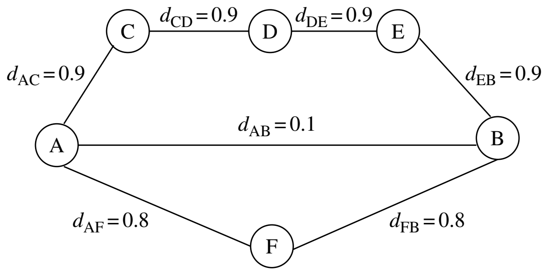


Figure 8.1 Illustration of the ETX routing metric

The direct transmission from A to B incurs 10 retransmissions on average. The long path through nodes C, D, and E incurs 1.11 retransmissions on each link, hence a total of 4.44 retransmissions. The third path, through node F, incurs 1.25 retransmissions on each link for a total of 2.5 retransmissions. This is the ETX-minimizing path. This example shows that ETX favors neither long paths involving a large number of short-distance (high-quality) links, nor very short paths involving a few long-distance (low-quality) links, but paths that are somewhere in between.

The ETX metric has many advantages. By minimizing the number of transmissions required, it improves bandwidth efficiency as well as energy efficiency. It also explicitly addresses link asymmetry by measuring reception probabilities in both directions.

One of the practical challenges is in determining the values of the packet reception probabilities d_f and d_r, by performing an appropriate link monitoring procedure. This can be done through some form of periodic measurement using sliding windows.

8.2.2 Metrics for energy–reliability tradeoffs (MOR/MER)

If the environment contains highly mobile objects, or if the nodes are themselves mobile, the quality of links may fluctuate quite rapidly. In this case, use of ETX-like metrics based on the periodic collection of packet reception rates may not be useful/feasible.

Reliable routing metrics for wireless networks with rapid link quality fluctuations have been derived analytically [106]. They explicitly model the wireless channel as having multi-path fading with Rayleigh statistics (fluctuating over time), and take an outage probability approach to reliability. Let d represent the distance between transmitter and receiver, η the path-loss exponent, SNR the normalized signal-to-noise ratio without fading, f the fading state of the channel, then the instantaneous capacity of the channel is described as:

$$C = \log\left(1 + \frac{|f|^2}{d^\eta} SNR\right) \tag{8.2}$$

The outage probability P_{out} is defined as the probability that the instantaneous capacity of the channel falls below the transmission rate R. It is shown that

$$P_{\text{out}} = 1 - \exp\left(\frac{-d^\eta}{\mu SNR^*}\right) \tag{8.3}$$

where $SNR^* = SNR/(2^R - 1)$ is a normalized SNR, and $\mu = E[|f|^2]$ is the mean of the Rayleigh fading. Based on this formulation, the authors derive the following results for the case that each transmission is limited to the same power:

Theorem 8

Let the end-to-end reliability of a given route be defined as the probability that none of the intermediate links suffer outage. Then, assuming that each link has the same transmitted signal-to-noise ratio, the most reliable route between two nodes is one that minimizes the path metric $\sum_i d_i^\eta$, where d_i is the distance of the ith *hop in the path.*

This metric (d^η for each link of distance d) is referred to as the minimum outage route (MOR) metric.

They also derive the following result for the case with power control:

Theorem 9

The minimum energy route between nodes that guarantees a minimum end-to-end reliability $R_{\min}$ is the route which minimizes the path metric $\sum_i \sqrt{d_i^\eta}$.

In this case the metric for each link is $\sqrt{d^\eta}$ with a proportional power setting, which is referred to as the minimum energy route (MER) metric. It should be noted, however, that here the energy that is being minimized is only the distance-dependent output power term – not the cost of receptions or other distance-independent electronics terms. It turns out that the MER metric can also be used to determine the route which maximizes the end-to-end reliability metric, subject to a total power constraint.

One key difference between MOR/MER metrics and the ETX metric is that they do not require the collection of link quality metrics (which can change quite rapidly in dynamic environments), but assume that the fading can be modelled by a Rayleigh distribution. Also, unlike ETX, this work does not take into account the use of acknowledgements.

8.3 Routing with diversity

Another set of techniques exploits the diversity provided by the fact that wireless transmissions are broadcast to multiple nodes.

8.3.1 Relay diversity

The authors of [106] also propose and analyze the reliability–energy tradeoffs for a simple technique for providing diversity in wireless routing that exploits the wireless broadcast advantage. This is illustrated in Figure 8.2 by a simple two-hop route from A to B to C. With traditional routing, the reliability of the

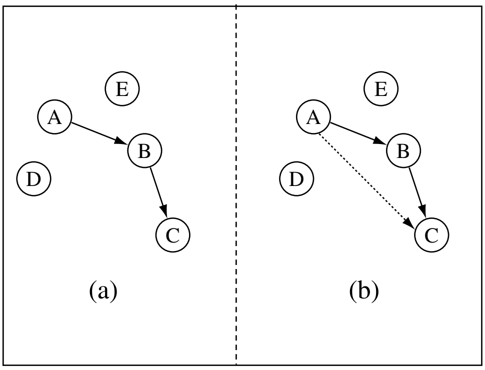


Figure 8.2 Illustration of relay diversity: (a) traditionally C receives a packet from A successfully only if the transmission from A to B and B to C are both successful; (b) with relay diversity, the transmission could also be successful in addition if the transmission from A to B is overheard successfully by C

path is purely a function of whether transmissions on A and B and B and C were both successful. However, if C is also allowed to accept packets directly from A (whenever they are received without error), then the reliability can be further increased without any additional energy expenditure.

Allowing such packet receptions within two hops, it is shown that in the high-SNR regime, the end-to-end outage probability decays as $(SNR)^{-2}$ (a second-order diversity gain). The authors further conjecture that, when nodes within L hops can communicate with each other with high SNR, the end-to-end outage probability would decay as SNR^{-L}.

One tradeoff in using this technique is that it requires a larger number of receivers to be actively overhearing each message, which may incur a radio energy penalty.

8.3.2 Extremely opportunistic routing (ExOR)

A related innovative network layer approach to robust routing that takes unique advantage of the broadcast wireless channel for diversity is the extremely opportunistic routing (ExOR) technique [10]. Unlike traditional routing techniques, in ExOR the identity of the node, which is to forward a packet, is not pre-determined before the packet is transmitted. Instead, it ensures that the node closest to the destination that receives a given packet will forward the packet further. While this technique does not explicitly use metric-based routing, the protocol is designed to minimize the number of transmissions as well as the end-to-end routing delay (Figure 8.3).

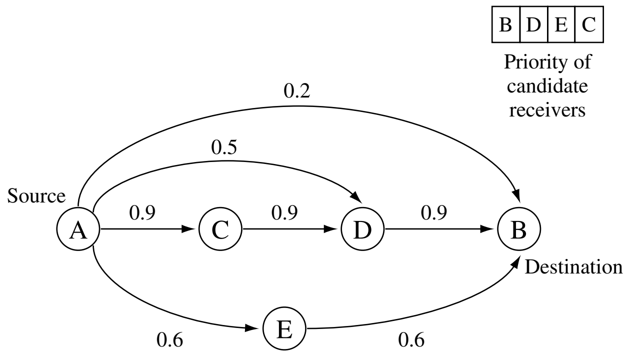


Figure 8.3 ExOR routing

In detail, the protocol has three component steps:

1. Priority ordering: At each step, the transmitter includes in the packet a schedule describing the candidate set of receivers and the priority order in which they should forward the packet. The candidate list and ordering are constructed to ensure high likelihood of forward progress, as well as to ensure that receivers that lie on shorter routes to the destination (which are also generally the closest to the destination) have higher priority.
2. Transmission acknowledgements: A distributed slotted MAC scheme is used whereby each candidate receiver sends the ID of the highest-priority successful recipient known to it. All nodes listen to all ACK slots. This constitutes a distributed election procedure whereby all nodes can determine which node has the highest priority among the candidates that received the packet successfully. Even if a node does not directly hear the highest-priority node's ACK, it is likely to hear that node's ID during another candidate's ACK. Thus, by transmitting the IDs as per this protocol, and having all nodes wait to receive all ACKs, an attempt is made to suppress duplicate forwarding to the greatest extent possible.
3. Forwarding decision: After listening to all ACKs, the nodes that have not heard any IDs with priorities greater than their own will transmit. There is a small possibility that ACK losses can cause duplicate transmissions; these are further minimized to some extent by ensuring that no node will ever retransmit the same packet twice (upon receiving it from different transmitters).

There are several nice features of the ExOR protocol. The chief among these is that it implements a unique network-layer diversity technique that takes advantage of the broadcast nature of the wireless channel. Nodes that are further away (yet closer to the destination) are less likely to receive a packet, but, whenever they do, they are in a position to act as forwarders. The almost counter-intuitive approach of routing without pre-specifying the forwarding node thus saves on expected delay as well as the number of transmissions.

As with the relay diversity technique described before, ExOR also requires a larger set of receivers to be active, which may have an energy penalty. Moreover, to determine the priority ordering of candidate receivers, the inter-node delivery ratios need to be tracked and maintained.

8.4 Multi-path routing

One basic solution for robustness that has been proposed with several variations is the use of multiple disjoint or partially disjoint routes to convey information from source to destination. There is considerable prior literature on multi-path routing techniques; we will highlight here only a few recent studies focused on sensor networks.

8.4.1 Braided multi-path routing

A good example of the use of localized enforcement-based routing schemes, such as Directed Diffusion to provide for multi-path robustness, is the "braided multi-path" schemes for alternate path routing in wireless sensor networks [58]. In alternate path routing, there is a primary path that is mainly used for routing and several alternate paths are maintained for use when a failure occurs on the primary path. A braided path is one where node disjointedness between the alternate paths is not a strict requirement. It is defined as one in which for each node on the main path there exists an alternate path from the source to the sink that does not contain that node, but which may otherwise overlap with the other nodes on the main path.

The braided multi-path construction is compared with a localized disjoint multi-path construction in the study [58] in terms of resiliency to both isolated and pattern failures, as well as the overhead required for multi-path maintenance (which is assumed to be proportional to the number of participating nodes). While the disjoint multi-path techniques have greater resiliency to pattern failures than the braided multi-path techniques (which suffer due to the geographic proximity of all alternate paths), this comes at the expense of significantly greater overhead. Particularly for isolated failures, the braided approach can be significantly more resilient and energy-efficient.

8.4.2 Gradient cost routing (GRAd)

The gradient routing technique (GRAd) [163], provides a simple mechanism for multi-path robustness. All nodes in the network maintain an estimated cost to each active destination (this set would be restricted to the sinks in a sensor network). In the simplest case, the cost metric would be just the number of hops; however, the protocol can be enhanced to handle other metrics. When a packet is transmitted, it includes a field that indicates the cost it has accrued to date (i.e. number of hops traversed), and a remaining value field, that acts as a TTL (time-to-live) field for the packet. Any receiver that receives this packet and notes that its own cost is smaller than the remaining value of the packet can forward the message, so long as it is not a duplicate. Before forwarding, the accrued cost field is incremented by one and the remaining value field is decremented by one (in the case of hop-count metric, the increment/decrements would be accordingly different for other metrics). One issue that needs to be taken into account in practice is that in the basic GRAd scheme cost fields are established using a reverse-path approach, which assumes the existence of bidirectional links. Since GRAd allows multiple nodes to forward the same message, it acts essentially as a limited directed flood and provides significant robustness, at the cost of larger overhead.

8.4.3 Gradient Broadcast routing (GRAB)

The Gradient Broadcast mechanism (GRAB) [236] enhances the GRAd approach by incorporating a tunable energy–robustness tradeoff through the use of credits. Similar to GRAd, GRAB also maintains a cost field through all nodes in the network. The packets travel from a source to the sink, with a credit value that is decremented at each step depending on the hop cost. An implicit credit-sharing mechanism ensures that earlier hops receive a larger share of the total credit in a packet, while the later hops receive a smaller share of the credit. An intermediate forwarding node with greater credit can consume a larger budget and send the packet to a larger set of forwarding eligible neighbors. This allows for greater spreading out of paths initially, while ensuring that the diverse paths converge to the sink location efficiently. This is illustrated in Figure 8.4, which shows the set of nodes that may be used for forwarding between a given source and the sink.

The GRAB-forwarding algorithm works as follows. Each packet contains three fields: (i) R_o – the credit assigned at the originating node; (ii) C_o – the cost-to-sink at the originating node; and (iii) U – the budget already consumed from the source to the current hop. The first two fields never change in the packet, while

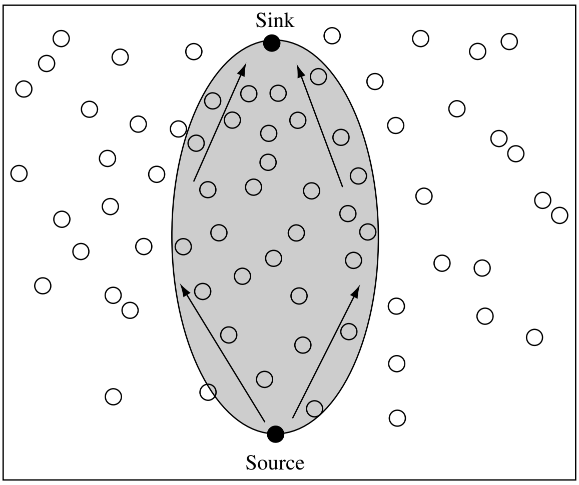


Figure 8.4 Forwarding mesh for gradient broadcast routing

the last is incremented at each step, depending on the cost of packet transmission (e.g., it could be related to the power setting of the transmission). To prevent routing loops, only receivers with lower costs can be candidates for forwarding. Each candidate receiver i with a cost-to-sink of C_i computes a metric called β and a threshold θ as follows:

$$\beta = 1 - \frac{R_{oi}}{R_o} \tag{8.4}$$

$$\theta = \left(\frac{C_i}{C_o}\right)^2 \tag{8.5}$$

where

$$R_{oi} = U - (C_o - C_i) \tag{8.6}$$

The expression R_{oi} determines how much credit has already been used up in traversing from the origin to the current node. The metric β, therefore, is an estimate of the remaining credit of the packet. The threshold θ is a measure of remaining distance to the sink. The candidate node will forward the message so long as $\beta > \theta$. The square gives the θ threshold a weighting, so that the threshold is more likely to be exceeded in the early hops than the later hops, as desired. The authors of GRAB show that the choice of initial credit, R_o, provides a tunable parameter to increase robustness at the expense of greater energy consumption.

8.5 Lifetime-maximizing energy-aware routing techniques

A number of studies have explored the issue of energy aware, lifetime-maximizing routing approaches for wireless ad hoc and sensor networks. Many of these are based on identifying and defining suitable shortest-path link metrics, while some derive energy-efficient routes for a network using a global optimization formulation.

8.5.1 Power-aware routing

In an ideal, lightly loaded environment, assuming all links require the same energy for the transmission of a packet, the traditional minimum hop-count routing approach will generally result in minimum energy expended per packet. If different links have uneven transmission costs, then the route that minimizes the energy expended in end-to-end delivery of a packet would be the shortest path route computed using the metric $T_{i,j}$, the transmission energy for each link i, j. However, in networks with heterogeneous energy levels, this may not be the best strategy to extend the network lifetime (defined, for instance, as the time till the first node exhaustion).

The basic power-aware routing scheme [200] selects routes in such a way as to prefer nodes with longer remaining battery lifetime as intermediate nodes. Specifically, let R_i be the remaining energy of an intermediate node i, then the link metric used is $c_{i,j} = \frac{1}{R_i}$. Thus, the path P (indicating the sequence of transmitting nodes for each hop) selected by a shortest-cost route determination algorithm (such as Dijkstra's or Bellman-Ford) would be one that minimizes $\sum_{i \in P} \frac{1}{R_i}$.

8.5.2 Lifetime-maximizing routing

While minimizing per-hop transmission costs minimizes total energy, avoiding nodes with low residual energy prevents early node failure. However, considering these goals separately, as in the above, may not optimize the system lifetime. What is needed is a technique that balances the two goals, selecting the minimum energy path when all nodes have high energy at the beginning, and avoiding the low residual energy nodes towards the end.

Chang and Tassiulas [20] propose the following link metric, which is a function of the transmission cost on the link $T_{i,j}$, the residual energy of the transmitting node R_i, and the initial energy of the transmitting node E_i:

$$c_{i,j} = T_{i,j}^{a} R_i^{-b} E_i^{c} \tag{8.7}$$

This general formulation captures a wide range of metrics. If $(a, b, c) = (0, 0, 0)$, we have a minimum hop metric; if $(a, b, c) = (1, 0, 0)$, we have the minimum energy-per-packet metric; if $b = c$, then normalized residual energies are used, while $c = 0$ implies that absolute residual energies are used; if $(a, b, c) = (0, 1, 0)$, we have the inverse-residual-energy metric suggested in [200]. However, simulation results in [20] suggest that a non-zero a and relatively large $b = c$ terms provide the best performance (e.g. (1, 50, 50)).

8.5.3 Load-balanced energy-aware routing

In order to provide an additional level of load balancing, which is necessary in a static network, the following cost-based probabilistic forwarding mechanism [189] can be used. Nodes forward packets only to neighbors that are closer to the destination. Let the cost to destination from node i through a candidate neighbor j be given as $C_{ij} = C_j + c_{ij}$, where C_j is the expected minimum cost-to-destination from j and c_{ij} is any link cost metric (e.g. the $T_{i,j}^a R_i^{-b} E_i^c$ metric discussed above). For each neighbor j in its set of candidate neighbors (N_i), node i assigns a forwarding probability that is inversely proportional to the cost to destination

$$P_{i,j} = \frac{C_{ij}^{-1}}{\sum_{k \in N_i} C_{ik}^{-1}} \tag{8.8}$$

Node i then calculates the expected minimum cost to destination for itself as

$$C_i = \sum_{j \in N_i} P_{ij} C_{ij} \tag{8.9}$$

Each time the node needs to route any packet, it forwards to any of its neighbors randomly with the corresponding probability. This provides for load balancing, preventing a single path from rapidly draining energy.

8.5.4 Flow optimization formulations

Chang and Tassiulas [21] also formulate the global problem of maximizing the network lifetime with known origin–destination flow requirements as a linear program (LP), and propose a flow augmentation heuristic technique based on iterated saturation of shortest-cost paths to solve it. The basic idea is that in each iteration every origin node computes the shortest cost path to its destination, and augments the flow on this path by a small step. After each step the costs are recalculated, and the process repeated until any node runs out of its initial energy E_i.

We should note that such LP formulations have been widely used by several authors in the literature to study performance bounds and derive optimal routes. Bhardwaj *et al.* use LP formulations to derive bounds on the lifetime of sensor networks [9]. LP-based flow augmentation techniques are used by Sadagopan and Krishnamachari [179] for a related problem involving the maximization of total data gathered for a finite energy budget. Techniques to convert multi-session flows obtained from such linear programming formulations into single-session flows are discussed in [151]. Kalpakis *et al.* [99] also present an integer flow formulation for maximum lifetime data-gathering with aggregation, along with near-optimal heuristics. Ordonez and Krishnamachari present non-linear flow optimization problems that arise when variable power-based rate control is considered, and compare the gains obtained in energy-efficient data-gathering with and without power control [110].

To illustrate this approach, consider the following simple flow-based linear program. Let there be n-numbered source nodes in the network, and a sink labelled $n+1$. Let f_{ij} be the data rate on the corresponding link, C_{ij} the cost of transmitting a bit on the link, R the reception cost per bit at any node, T the total time of operation under consideration, E_i the available energy at each node, and B_i the total bandwidth available at each node.

$$\max \quad \sum_{j=i}^{n} f_{i,n+1} \cdot T$$

$$\text{s.t. } \forall i \neq (n+1), \sum_{j=1}^{n+1} f_{ij} - \sum_{j=1}^{n} f_{ji} \geq 0 \qquad \text{(a)}$$

$$\left(\sum_{j=1}^{n+1} f_{ij} \cdot C_{ij} + \sum_{j=1}^{n} f_{ji} \cdot R \right) \cdot T \leq E_i \qquad \text{(b)}$$

$$\sum_{j=1}^{n+1} f_{ij} + \sum_{j=1}^{c} f_{ji} \leq B_i \qquad \text{(c)}$$

This linear program maximizes the total data gathered during the time duration T. It incorporates (a) a flow conservation constraint, (b) a per-node energy constraint, and (c) a shared bandwidth constraint.

8.6 Geographic routing

Since location information is often available to all nodes in a sensor network (if not directly, then through a network localization algorithm) in order to provide location-stamped data or satisfy location-based queries, geographic routing techniques [135] are often a natural choice.

8.6.1 Local position-based forwarding

The simplest version of geographic forwarding, described by Finn [53], is to forward the packet at each step to the neighbor that is closest to the destination (the location or ID of this neighbor is marked in the packet, based on neighborhood information obtained through a beaconing process). There are several variants of this greedy forwarding mechanism. In the most forward within R strategy (MFR) [208], the packet is forwarded to the neighbor whose projection on the line joining the current node and the destination is the farthest (note that this is not always the same as the greedy forwarding). In another variant, the packet is forwarded to the nearest neighbor with forward progress (NFP) [85], so that contention can be minimized and the wireless link quality is high. Yet another variant is the random-forwarding technique in which the packet is forwarded at random to a neighbor so long as it makes forward progress. We should also note that the ExOR technique, described above, is a form of implicit greedy geographic forwarding, which does not require beaconing – the timers are designed so that packets are likely to be forwarded (only) by the node closest to the destination that receives the packet correctly.

8.6.2 Perimeter routing (GFG/GPSR)

One major shortcoming of the greedy forwarding technique is that it is possible for it to get stuck in local maxima/dead-ends. Such dead-ends occur (see Figure 8.5) when a node has no neighbors that are closer to the destination than itself. For this case, the greedy-face-greedy (GFG) algorithm [14], which is the basis of the greedy perimeter stateless routing protocol (GPSR) [102], routes along the face of a planar sub-graph using the right-hand rule. The planar sub-graph can be obtained using localized constructions such as the Gabriel graph and the relative neighbor graph constructions. A packet switches from greedy to face-routing mode whenever it reaches a dead end, is then routed using face routing, and then reverts back to greedy mode as soon as it reaches a node that is closer to the destination than the dead-end node. Other studies have examined ways to improve upon and get provable efficiency guarantees with face-routing approaches. It should be kept in mind, however, that the likelihood that such dead ends exist decreases with network density; it can be shown that, if the graph is dense enough that each interior node has a neighbor in every $2\pi/3$ angular sector, then greedy forwarding will always succeed [49].

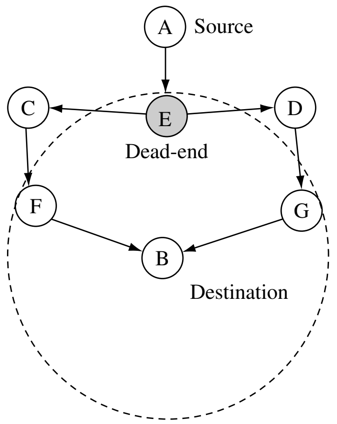


Figure 8.5 A dead-end for greedy geographic forwarding

8.6.3 The PRR*D metric

Seada *et al.* show in [187] that greedy geographic forwarding techniques exhibit a distance-hop–energy tradeoff over real wireless links. If at each step the packet travels a long distance (as in the basic greedy and MFR techniques), the total number of hops is minimized; however, because of the distance, each hop is likely to be a weak link, with poor reception rate requiring multiple transmissions for successful delivery. At the other extreme, if the packet travels only a short distance at each hop (as with the NFP technique), the links are likely to be good quality; however there are multiple hops to traverse, which would increase the number of transmissions as well. By extensive simulations, real experiments as well as analysis, it is shown that the localized metric that maximizes the energy efficiency while providing a good end-to-end delivery reliability is the product of the packet reception rate on the link (probability of successful delivery) and the distance improvement towards the destination D, which is known as the $PRR * D$ metric. Thus, when it has a packet to transmit to a given destination, each node selects the neighbor with the highest $PRR * D$ metric to be the one to forward the message further.

8.6.4 Geographical energy-aware routing (GEAR)

The geographical and energy-aware routing mechanism (GEAR) [240] is designed to propagate queries/interests from a sink to all sensor nodes in a geographically scoped destination region (e.g. a square sub-area with specified bounds). The packets are routed from the origin to the destined region through a series of forwarding steps, which make use of an adaptive learned cost function for each neighbor that is modified over time to provide a balance between reachability and energy efficiency. This approach also provides robustness to dead ends. When the packet reaches the destination region, a recursive forwarding technique is employed as follows. The region is split into *k* sub-regions, and a packet is forwarded to each sub-region. This split–forward sequence is repeated until the region contains only one node, at which point the packet has been successfully propagated to all nodes within the query region.

Finally, we should mention in this section the trajectory-based forwarding technique (TBF) [148], which is also an important geographic routing technique for sensor networks. As a significant application of TBF applies to the routing of queries, however, we shall defer its description to the next chapter.

8.7 Routing to mobile sinks

Mobility of nodes in the network adds a significant challenge. The study of routing over mobile ad hoc networks (MANET) has indeed been an entire field in itself, with many protocols such as DSR, AODV, ZRP, ABR, TORA, etc. proposed to provide robustness in the face of changing topologies [158, 211]. A thorough treatment of networking between arbitrary end-to-end hosts in the case where all nodes are mobile is beyond the scope of this text. However, even in predominantly static sensor networks, it is possible to have a few mobile nodes. One scenario in particular that has received attention, is that of *mobile sinks*. In a sensor network with a mobile sink (e.g. controlled robots or humans/vehicles with gateway devices), the data must be routed from the static sensor sources to the moving entity, which may not necessarily have a predictable/deterministic trajectory. A key advantage of incorporating mobile sinks into the design of a sensor network is that it may enable the gathering of timely information from remote deployments, and may also potentially improve energy efficiency.

We describe below a few studies that propose solutions to different variants of this problem.

8.7.1 Two-tier data dissemination (TTDD)

In the two-tier data dissemination (TTDD) approach [239], all nodes in the network are static, except for the sinks that are assumed to be mobile with unknown/uncontrolled mobility. The data about each event are assumed to originate from a single source. Each active source creates a grid overlay dissemination network over the static network, with grid points acting as dissemination nodes (see Figure 8.6). A mobile sink, when it issues queries for information, sends out a locally controlled flood that discovers its nearest dissemination point. The query is then routed to the source through the overlay network. The sink includes in the query packet information about its nearest static neighbor, which acts as a *primary agent*. An alternative *immediate agent* is also chosen when the sink is about to go out of reach of the primary agent for robust delivery. The source sends data to the sink through the overlay dissemination network to its closest grid dissemination node, which then forwards it to its primary agent. As the sink moves through the network, new primary agents are selected and the old ones time out; when a sink moves out of reach of its nearest dissemination node, a new dissemination node is discovered and the process continues.

8.7.2 Asynchronous dissemination to mobile sinks (SEAD)

The scalable energy-efficient asynchronous dissemination technique (SEAD) presented in [107] provides for communication of information from a given source in a static sensor network to multiple mobile sinks. Each mobile sink selects a nearby static access node to communicate information to and from the source. Only the access node keeps track of sink movement, so long as it does not move too far away. When the hop-count between the sink and the nearest access point exceeds a threshold, a new access node is selected by the sink. Data are sent from the source first to the various access nodes through a dynamically

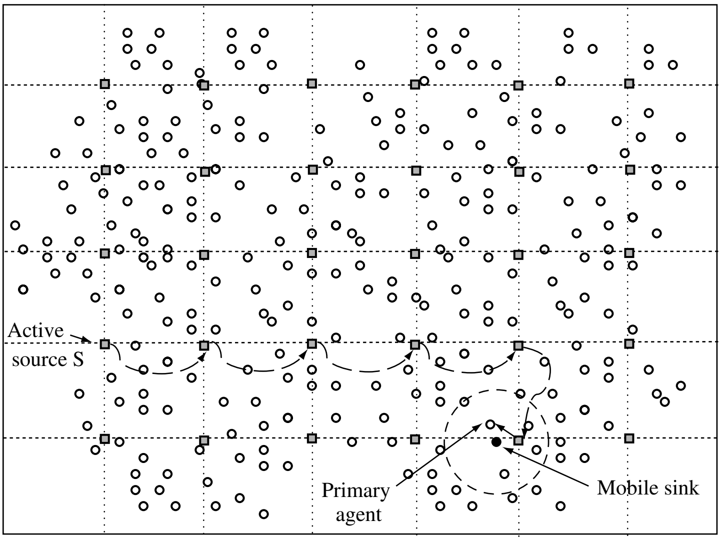


Figure 8.6 The two-tier data dissemination technique

maintained multi-cast dissemination tree, which then forwards the information to their respective sinks. The multi-cast tree construction scheme described for SEAD relies on the replication of source data at multiple points in the network. Each access node selects a nearby replica to obtain data from, based on a minimum-cost increase criterion.

8.7.3 Data mules

For sparsely deployed sensor networks (e.g. deployed over large areas), the network may never be truly connected; in the most extreme case no two sensor devices may be within radio range of each other. The MULE (mobile ubiquitous LAN extensions) architecture [190] aims to provide connectivity in this environment through the use of mobile nodes that may help transfer data between sensors and static access points, or may themselves act as mobile sinks.

It is assumed that MULE nodes do not have controlled mobility and that their movements are random and unpredictable. Whenever a MULE node comes into contact with a sensor node, it is assumed that all the sensors data are transferred over to the MULE. Whenever the MULE comes into contact with an access point, it transfers all information to the access point. It is assumed that there is sufficient density of MULE nodes, with sufficiently uniform mobility, so that all sensor nodes can be served, although delays are likely to be quite high. Both MULEs and sensors are assumed to have limited buffers, so that new data are stored or transferred only if there is buffer space available.

The MULE architecture has been analyzed using random walks on a grid [190]. The analysis provides an insight into the scaling behavior of this system with respect to number of sensors, MULEs, and access points. One conclusion of the study worth noting is that the buffer size of MULE nodes can be increased to compensate for fewer MULE nodes in terms of delivery rates (albeit at the expense of increased latency), but increasing sensor buffers alone does not necessarily have a similar effect.

8.7.4 Learning enforced time domain routing

The hybrid learning enforced time domain routing (HLETDR) technique (Figure 8.7) is proposed for situations where the sink follows a somewhat predictable but stochastic repeated pattern [7]. It is assumed that events happen rarely, so that the sink does not issue queries, but rather that sources need to push data about events towards the sink. Nodes near the path of the sink's movements, called moles, *learn* the probability distribution of the sink's appearance in their

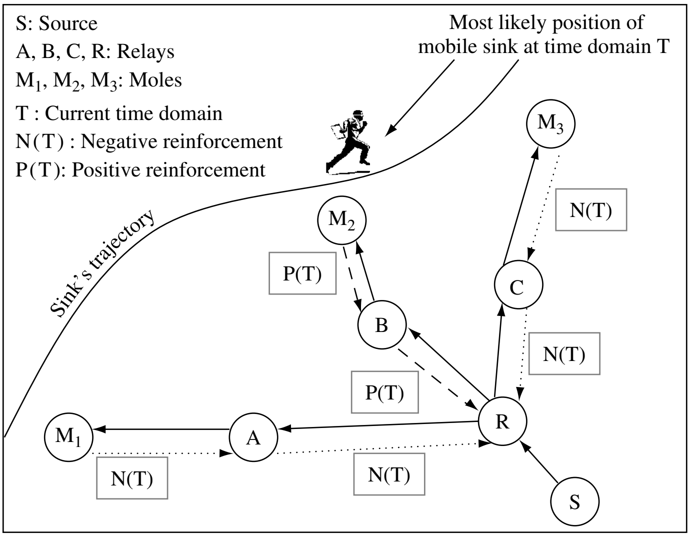

S: Source
A, B, C, R: Relays
M_1, M_2, M_3: Moles
T : Current time domain
N(T) : Negative reinforcement
P(T): Positive reinforcement
Most likely position of mobile sink at time domain T
Sink's trajectory
M_1
M_2
M_3
A
B
C
R
S
P(T)
P(T)
N(T)
N(T)
N(T)
N(T)

Figure 8.7 Illustration of hybrid learning enforced time domain routing

vicinity. The periodic time between tours of the sink is divided into multiple domains, such that the sink may be more likely to be in the vicinity of one set of moles in one time domain, and in the vicinity of another set of moles in another time domain.

For each time domain, local forwarding probabilities are maintained at intermediate nodes. When data are generated, depending on the time, they are routed through the intermediate nodes based on these probabilities to try and reach a mole that the sink will pass by. Initially, the probability weights at nodes are all equal, resulting in unbiased random walks. Over time, these weights are reinforced positively or negatively by moles, depending on the sink probability distribution and success of the data delivery. Multiplicative weight update rules for reinforcements are found to be most efficient and robust. A few iterations may suffice to determine efficient routes for data to reach a mole that is highly likely to encounter the sink and be delivered successfully.

8.8 Summary

We have examined a number of issues and design concepts relevant to reliable, energy-efficient routing in wireless sensor networks: selection of routing metrics, multi-path routing, geographic routing, and delivery of data to mobile nodes.

Reliability in routing can be achieved in many ways. One approach is metric based. The ETX metric requires nodes in the network to monitor the quality of links to their neighbors to find routes that minimize the number of transmissions incurred due to ARQ retries. The MOR/MER metrics, developed analytically, are better suited for rapidly time-varying channels and offer an alternative way to effect energy–reliability tradeoffs. Another approach to reliable data delivery is the use of multi-path routing techniques, such as the braided multi-path mechanism, and the GRAd, GRAB routing protocols. Finally, reliability can be enhanced even at the network layer by exploiting the spatial diversity of independent fading channels, as in the relay diversity technique and the extremely opportunistic routing mechanism.

Energy-aware routing techniques utilize metrics that take into account the residual lifetimes of intermediate nodes on the routing path. They can be enhanced with probabilistic forwarding to provide some degree of energy–load balancing. Global solutions in the form of flow-based optimization formulations have also been used to analyze lifetime-maximizing flows and develop distributed algorithms for energy-efficient routing.

Given that nodes in a sensor network are likely to have at least coarse-grained position information, geographic forwarding techniques, which provide low-overhead stateless routing, can be an attractive option.

Finally, while our focus has largely been on sensor networks with static nodes, in some applications it may be necessary to include limited mobility in the form of a moving sink. Routing techniques such as TTDD, SEAD, and HLETDR address this scenario.

Exercises

8.1 *ETX:* For the directed graph labelled with reception probabilities shown in Figure 8.3 (ignore the priorities associated with ExOR, and assume $d_r = 1$ on all links), determine the optimal ETX route from node A to node B.

8.2 *Relay diversity and MAC:* Explain why the relay diversity scheme may not work well with some sleep-oriented MAC protocols proposed for sensor networks.

8.3 *Relay diversity:* For the example of relay diversity shown in Figure 8.2, say the probabilities of reception for the links A–B, B–C, and A–C were 0.8, 0.8, and 0.6 respectively. What is the probability of successful reception at C without and with relay diversity?

8.4 *Timer-based ExOR routing:* Consider a variant of ExOR routing in which successful recipients of a message set a timer for retransmission of the message depending upon their priority. Thus, in the example of Figure 8.3, node B's timer would be set to 1, node D to 2, node E to 3, node C to 4. If a node hears another one forwarding the message, it cancels its timer. After a node's timer expires, it will forward the message itself. In this case, once A sends the packet, what is the expected delay before the first of its recipients forwards the message?

8.5 *Flow formulation:* Adapt the linear program given in Section 8.5.4 for a fairness-oriented objective function that maximizes the minimum flow rate from all sources.

8.6 Greedy geographic routing fails when a forwarding node on the path finds no neighbors within range that are closer than itself to the destination. Prove that this implies the existence of a $2\pi/3$ angular sector centered at this node in which it has no neighbors.

8.7 *MULE simulation study:* On a 10×10 square grid, place the sink node at the bottom-left-most grid, and ten sources at random squares. Simulate the movement of k MULE nodes (with varying k), such that all execute independent unbiased random walks on the grid, moving to a neighboring cell at each time step. Assume the MULE nodes pick up one unit of information from the source when they are in the same grid, and drop off all information they contain when they arrive at the sink. Assume that the sources always have data available for pick-up and an infinite buffer, and that MULEs have an infinite buffer too. Analyze significant metrics, such as the average time delay between each visit to the sink, average size of the MULE buffers, average throughput between sources and the sink, as functions of the number of MULE nodes. What is the impact of placing additional static sink nodes?

9

Data-centric networking

9.1 Overview

A fundamental innovation in the area of wireless sensor networks has been the concept of *data-centric networking*. In a nutshell, the idea is this: routing, storage, and querying techniques for sensor networks can all be made more efficient if communication is based directly on application-specific data content instead of the traditional IP-style addressing [74].

Consider the World Wide Web. When one searches for information on a popular search site, it is possible to enter a query directly for the content of interest, find a hit, and then click to view that content. While this process is quite fast, it does involve several levels of indirection and names: ranging from high-level names, like the query string itself, to domain names, to IP (internet protocol) addresses, and MAC addresses. The routing mechanism that supports the whole search process is based on the hierarchical IP addressing scheme, and does not directly take into account the content that is being requested. This is advantageous because the IP is designed to support a huge range of applications, not just web searching. This comes with increased indirection overhead in the form of the communication and processing necessary for binding; for instance the search engine must go through the index to return web page location names as the response to the query string, and the domain names must be translated to IP addresses through DNS. This tradeoff is still quite acceptable, since the Internet is not resource constrained.

Wireless sensor networks, however, are qualitatively different. They are application specific so that the data content that can be provided by the sensors is relatively well defined *a priori*. It is therefore possible to implement network operations (which are all restricted to querying and transport of raw and processed sensor data and events) directly in terms of named content. This data-centric approach to networking has two great advantages in terms of efficiency:

1. Communication overhead for binding, which could cause significant energy wastage, is minimized.
2. In-network processing is enabled because the content moving through the network is identifiable by intermediate nodes. This allows further energy savings through data aggregation and compression.

9.2 Data-centric routing

9.2.1 Directed diffusion

One of the first proposed *event-based* data-centric routing protocols for WSN is the directed diffusion technique (Figure 9.1) [96, 97].

This protocol uses simple attribute-based naming as the fundamental building block. Both requests for information (called *interests*) and the notifications of observed events are described through sets of attribute–value pairs. Thus, a request for 10 seconds worth of data from temperature sensors within a particular rectangular region may be expressed as follows:

```
type     = temperature     // type of sensor data
start    = 01:00:00        // starting time
interval = 1s             // once every second
duration = 10s            // for ten seconds
location = [24, 48, 36, 40] // within this region
```

And one of the data responses from a particular node may be:

```
type      = temperature // type of sensor data
value     = 38.3         // temperature reading
timestamp = 01:02:00    // time stamp
location  = [30, 38]    // x,y coordinate
```

The steps of the basic directed diffusion are as follows:

1. The sink issues an interest for a specific type of information that is flooded throughout the network (the overhead of this can be reduced if necessary by using geographic scoping or some other optimization). The interest may be periodically repeated if robustness is called for.
2. Every node in the network caches the interest while it is valid, and creates a local gradient entry towards the neighboring node(s) from which it heard

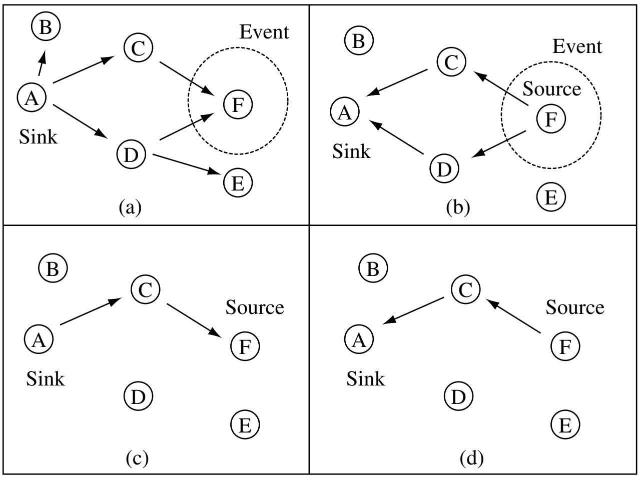


Figure 9.1 Directed diffusion routing

the interest. The sink's ID/network address is not available and hence not recorded, however the local neighbors are assumed to be uniquely identifiable through some link-layer address. The gradient also specifies a value (which could be an event rate, for instance).

3. A node which obtains sensor data that matches the interest begins sending its data to all neighbors it has gradients toward. If the gradient values stand for event rates then the rate to each neighbor must satisfy the gradients on the respective link. All received data are cached in intermediate nodes to prevent routing loops.
4. Once the sink starts receiving response data to its interest from multiple neighbors, it begins reinforcing one particular neighbor (or *k* neighbors, in case multi-path routing is desired), requesting it to increase the gradient value (event rate). These reinforcements are propagated hop by hop back to the source. The determination of which neighbor to reinforce can take into account other considerations such as delay, link quality, etc. Nodes continue to send data along the outgoing gradients, depending on their values.
5. (Optional) Negative reinforcements are used for adaptability. If a reinforced link is no longer useful/efficient, then negative reinforcements are sent to reduce the gradient (rate) on that link. The negative reinforcements could be implemented by timing out existing gradients, or by re-sending interests with a lower gradient value.

Essentially what directed diffusion does is (a) the sink lets all nodes in the network know what the sink is looking for, (b) those with corresponding data respond by sending their information through multiple paths, and (c) these are pruned via reinforcement so that an efficient routing path is obtained.

The directed diffusion mechanism presented here is highly versatile. It can be extended easily to provide multi-path routing (by changing the number of reinforced neighbors) as well as routing with multiple sinks/sources. It also allows for data aggregation, as the data arriving at any intermediate node from multiple sources can be processed/combined together if they correspond to the same interest.

9.2.2 Pull versus push diffusion

The basic version of directed diffusion described above can be viewed as a *two-phase pull* mechanism. In phase 1, the sink *pulls* for information from sources with relevant information by propagating the interest, and sources respond along multiple paths; in phase 2, the sink initiates reinforcement, then sources continue data transfer over the reinforced path. Other variants of directed diffusion include the *one-phase pull* and the *push diffusion* mechanisms [75].

The two-phase pull diffusion can be simplified to a one-phase pull mechanism by eliminating the reinforcements as a separate phase. In one-phase pull diffusion, the sink propagates the interest along multiple paths, and the matching source directly picks the best of its gradient links to send data and so on up the reverse path back to the sink. While potentially more efficient than two phase pull, this reverse-path selection assumes some form of bidirectionality in the links, or sufficient knowledge of the link qualities/properties in each direction.

In push diffusion, the sink does not issue its interests. Instead sources with event detections send exploratory data through the network along multiple paths. The sink, if it has a corresponding interest, reinforces one of these paths and the data-forwarding path is thus established.

The push and pull variants of diffusion have been compared and analyzed empirically [75] as well as through a simple mathematical model [111]. The results quantify the intuition that the pull and push variants are each appropriate for different kinds of applications. In terms of the route setup overhead, pull diffusion is more energy-efficient than push diffusion whenever there are many sources that are highly active generating data but there are few, infrequently interested sinks; while push diffusion is more energy-efficient whenever there are few infrequently active sources but there are many frequently interested sinks.

The threshold-sensitive energy-efficient sensor network protocol (TEEN) [132] is another example of a push-based data-centric protocol. In TEEN, nodes react immediately to drastic changes in the value of a sensed attribute and when this change exceeds a given threshold communicate their value to a cluster-head for forwarding to the sink.

9.3 Data-gathering with compression

Several researchers have investigated the combination of gathering information in a WSN by combining routing with in-network compression. While the exact type of compression involved can be quite application specific, these studies reveal a number of general principles and the tradeoffs involved. In most of these studies, the efficiency metric of interest is the total number of data bit transmissions (i.e. cumulative number of bits over each hop of transmission) per round of data-gathering from all sources. Besides providing energy efficiency by reducing the amount of transmissions, combining routing with compression also has the potential to improve network data throughput in the face of bandwidth constraints [186]. We now describe some of the pertinent techniques and analytical studies.

9.3.1 LEACH

The LEACH protocol [76] is a simple routing mechanism proposed for continuous data-gathering applications. In LEACH, illustrated in Figure 9.2, the

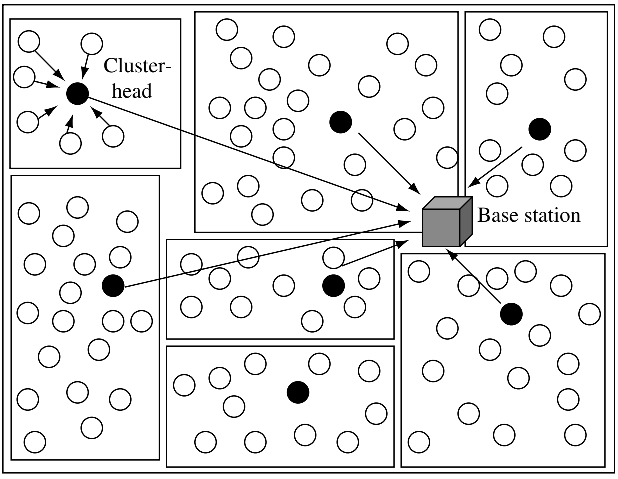


Figure 9.2 The LEACH cluster-based routing technique

network is organized into clusters. The cluster-heads periodically collect and aggregate/compress the data from nodes within the cluster using TDMA, before sending them to the sink. The cluster-heads may send to the sink through a direct transmission or through multiple hops. Cluster-heads are rotated periodically for load balancing.

9.3.2 Distributed source coding

There exist distributed source coding techniques based on the Slepian–Wolf theorem [34] that allow joint coding of correlated data from multiple sources without explicit communication, so long as the individual source rates satisfy certain constraints pertaining to different conditional entropies. These techniques require that the correlation structure be available in advance at the independent coders. While this approach increases the complexity of the coding and requires upfront collection of information about joint statistics (which may not be feasible in all practical conditions), it effectively makes routing and coding decisions independent of each other, since the independently coded data can be sent along the shortest paths to the sink. In a networked context, then, the only design consideration for energy efficiency is to ensure that sources near the destination are allocated higher rates [35].

9.3.3 Impact of compression

The gains due to in-network compression can be best demonstrated in the extreme case where the data from any number of sources can be combined into a single packet (e.g. duplicate suppression, when the sources generate identical data). In this case, if there are k sources, all located close to each other and far from the sink, then a route that combines their information close to the sources can achieve a k-fold reduction in transmissions, as compared with each node sending its information separately without compression. In general, the optimal joint routing–compression structure for this case is a minimum Steiner tree construction problem, which is known to be NP-hard. However there exist polynomial solutions for special cases where the sources are close to each other [109].

9.3.4 Network correlated data-gathering

Cristescu, Beferull–Lozano, and Vetterli [35] consider the case where all nodes are sources but the level of correlation can vary. In this case, when the data are completely uncorrelated then the shortest path tree provides the best solution (in minimizing the total transmission cost). However, the general case is treated by choosing a particular correlation model that preserves the complexity; in this model, only nodes at the leaf of the tree need to provide R bits, but all other interior nodes, which have side information from other nodes, need only generate r bits of additional information. The quantity $\rho = 1 - \frac{r}{R}$ is referred to as the correlation coefficient. Now, it can be shown that a travelling salesman path (that visits all nodes exactly once) provides an arbitrarily more efficient solution compared with shortest path trees as ρ increases. It is shown that this problem is NP-hard for arbitrary ρ vlaues.

A good approximation solution for the problem is the following combination of SPT and the travelling salesman paths. All nodes within some range of the sink (larger the ρ, smaller this range) are connected through shortest path trees, and beyond that each strand of the SPT is successively grown by adding nearby nodes, an approximate way to construct the travelling salesman paths. Thus the data from distant nodes are compressed sequentially up to a point, and then sent to the sink using shortest paths.

9.3.5 Simultaneous optimization for concave costs

Goel and Estrin [64] treat the case when the exact reduction in data that can be obtained by compressing k sources is not known. The only assumption that is made is that the amount of compression is concave with respect to k. This is a very reasonable assumption, as it essentially establishes a notion of monotonically diminishing contributions to the total non-redundant information; it simply means that the additional non-redundant information in the $j+1$th source is smaller than that of the jth source. A random tree construction is developed for this problem that ensures that the expected cost is within a factor $O(\log k)$ of the optimal, regardless of what the exact concave compression function is.

9.3.6 Scale-free aggregation

In practice the degree of spatial correlation is a function of distance, and better approximations are possible by taking this into account. Nodes nearby are able to provide higher compression than nodes at a greater distance. A model for the spatial correlations that captures this notion is considered [45]. Each node in a square grid of sensors is assumed to have information about the readings of all nodes within a k-hop radius. Nodes can communicate with any of their four cardinal neighbors. The aggregation/compression function considered is such that any redundant readings are suppressed in the intermediate hops. This work assumes a square grid network in which the source is located at the origin, on the bottom-left corner. The routing technique proposed is a randomized one: a node at location (x, y) forwards its data, after combining with any other preceding sources sending data through it, with probability $x/(x+y)$ to its left neighbor and with probability $y/x+y$ to its bottom neighbor. It is shown that this randomized routing technique can provide a constant factor approximation in expectation to the optimal solution.

9.3.7 Impact of spatial correlations on routing with compression

In [153], an empirically derived approximation is used to quantify spatial correlation in terms of joint entropies. The total joint information generated by an arbitrary set of nodes is obtained using an approximate incremental construction. At each step of this construction, the next nearest node that is at a minimum distance $d_{\min}$ to the current set of nodes is considered. This node contributes an amount of uncorrelated data equal to $\frac{d_{\min}}{c+d_{\min}} \cdot H_1$, where H_1 is the entropy of a single source and c a constant that characterizes the degree of spatial correlation. In the simplest case when all nodes are located on a line with equal spacing of d, this procedure yields the following expression for the joint entropy of n nodes:

$$H_n = H_1 + (n-1)\frac{d}{c+d} \cdot H_1 \tag{9.1}$$

Consider first the two extremes: (i) when $c = 0$, $H_n = nH_1$, the nodes are completely uncorrelated; (ii) when $c \to \infty$, $H_n = H_1$, the nodes are completely correlated.

Under this model, it becomes easy to quantify the total transmission cost of any tree structure where routing is combined with en route compression. An example scenario is considered with a linear set of sources at one end of a square grid communicating their data to a sink at the other end. An idealized distributed source coding is used as a lower bound for the communication costs in this setting. It is shown that at one extreme, when the data are completely uncorrelated ($c = 0$), the best solution is that of shortest path routing (since there is no possible benefit due to compression). At the other extreme, when data are perfectly correlated ($c \to \infty$), the best solution is that of routing the data among the sources first so that they are all compressed, before sending the combined information directly to the sink. For in-between scenarios, a clustering strategy is advocated such that the data from s nearby sources are first compressed together, then routed to the sink along the shortest path. It is shown that there is an optimal cluster size corresponding to each value of the correlation parameter. The higher the level of correlation, the larger the optimal cluster size. However, surprisingly, it is also found that there exists a near-optimal cluster size that depends on the topology and sink placement but is insensitive to the exact correlation level.

This result has a practical implication, because it suggests that a LEACH-like clustering strategy combined with compression at the cluster-heads can provide an efficient solution even in the case of heterogeneous or time-varying correlations.

9.3.8 Prediction-based compression

Another approach to combining routing and compression is to perform prediction-based monitoring [63]. The essence of this idea is that the base station (or a cluster-head for a region of the network) periodically gathers data from all nodes in the network, and uses them to make a prediction for data to be generated until the next period. In the simplest case, the prediction may simply be that the data do not change. More sophisticated predictions may indicate how the data will change over time (e.g. the predictions may be based on the expected movement trajectory of a target node, or in the case of diffuse phenomena such as heat and chemical plumes, these predictions may even be based on partial differential equations with known or estimated parameters [176]). This prediction is then broadcast to all nodes within the region. During the rest of the period, the component nodes only transmit information to the base station if their measurements differ from the predicted measurements.

9.3.9 Distributed regression

A closely related technique is the use of a distributed regression framework for model-based compression [69]. In this approach, nodes collaboratively determine the parameters of a globally parameterized function using local measurements. The model used is a weighted sum of local basis functions (which could just be polynomials, for example). Distributed kernel linear regression techniques are then used to determine the parameters. Compression is achieved by transmitting only the model parameters instead of the full data.

9.4 Querying

In basic data-gathering scenarios, such as those discussed above in connection with compression, information from all nodes needs to be provided continuously to the sink. In many other settings, the sink may not be interested in all the information that is sensed within the network. In such cases, the nodes may store the sensed information locally and only transmit it in response to a query issued by the sink. Therefore the querying of sensors for desired information is a fundamental networking operation in WSN. Queries can be classified in many ways:

1. **Continuous versus one-shot queries**: depending on whether the queries are requesting a long duration flow or a single datum.
2. **Simple versus complex queries**: complex queries are combinations that consist of multiple simple sub-queries (e.g. queries for a single attribute type); e.g. "What are the location and temperature readings in those nodes in the network where (a) the light intensity is at least w and the humidity level is between x and y OR (b) the light intensity is at least z." Complex queries may also be aggregate queries that require the aggregation of information from several sources; e.g. "report the average temperature reading from all nodes in region R1."
3. **Queries for replicated versus queries for unique data**: depending on whether the queries can be satisfied at multiple nodes in the network or only at one such node.
4. **Queries for historic versus current/future data**: depending on whether the data being queried for were obtained in the past and stored (either locally at the same node or elsewhere in the network), or whether the query is for current/future data. In the latter case data do not need to be retrieved from storage.

When the queries are for truly long-term continuous flows, the cost of the initial querying may be relatively insignificant, even if that takes place through naive flooding (as for instance, with the basic directed diffusion). However, when they are for one-shot data, the costs and overheads of flooding can be prohibitively expensive. Similarly, if the queries are for replicated data, a flooding may return multiple responses when only one is necessary. Thus other alternatives to flooding-based queries (FBQ) are clearly desirable.

9.4.1 Expanding ring search

One option is the use of an expanding ring search, illustrated in Figure 9.3. An expanding ring search proceeds as a sequence of controlled floods, with the radius of the flood (i.e. the maximum hop-count of the flooded packet) increasing at each step if the query has not been resolved at the previous step. The choice of the number of hops to search at each step is a design parameter that can be optimized to minimize the expected search cost using a dynamic programming technique [22].

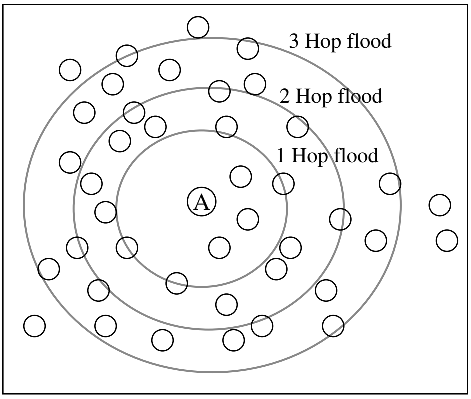


Figure 9.3 Expanding ring search

However, in the absence of replicated/cached information in the network, for arbitrarily located data, expanding ring searches do not provide useful gains over flooding. This is demonstrated in [24], which shows that less than 10% energy savings are obtained from expanding ring search compared with flooding when there is no caching/replication of data in the network, while the delay increases significantly. Intuitively, when there is replicated information, expanding ring searches are more likely to offer significant improvements, because the likelihood of resolving the query earlier would be higher. However, in this situation, other approaches may potentially provide better improvements.

9.4.2 Information-driven sensor query routing (IDSQ)

The information-driven sensor querying approach (IDSQ) [28] suggests an incremental approach to sensor tasking that is suitable for resource constrained, dynamic environments. The problem of how to route the query to a node with the maximum information gain is a core problem, that is addressed by the constrained anisotropic diffusion routing (CADR) technique. CADR essentially routes the query through a greedy search, making a sequence of local decisions at intermediate steps, based on sensor values of neighboring nodes. A composite objective function that combines the information utility and communication costs is first defined. These decisions can be made in different ways:

- by forwarding the query to the neighboring node with the highest objective function;
- by forwarding the query to the neighboring node with the steepest (local) gradient in the objective function;

- by forwarding the query to the neighboring node which maximizes a combination of the local gradient of the objective function and the distance improvement to the estimated optimum location (the information gain formulation used in IDSQ allows an estimation of the geographic location of the query destination).

One advantage of this approach, which provides a greedy descent towards the query destination, is that, if partial solutions are shipped back to the query-originating node, it is provided with incrementally better information as the query moves towards the global optimum. Further, depending on how the objective function is designed, this technique can minimize the energy needed to route the query to the destination by choosing the shortest path, or it can maximize the information gain by taking an irregular walk with more steps.

A related work on multi-step information-driven querying [122] aims to provide a minimum hop path through the network that maximizes the accumulated information gain. The following approach provides a solution: for each node i assign all links going into that node a cost of $L - u_i$, where L is a sufficiently large number and u_i the information utility at node i. Then find a shortest path from the origin to destination using this link metric (e.g., using Dijkstra's algorithm). This choice of cost function minimizes the number of hops, because of the large additive L terms. Among the minimum hop paths, the algorithm also maximizes the accumulated utility, because of the $-u_i$ terms in the minimization expression.

9.4.3 Fingerprint gradients (RUGGED)

The technique of query routing using fingerprint gradients (RUGGED) proposed by Faruque and Helmy [51] also makes use of sensor readings within the network to send the query to the node with the highest reading, which is assumed to be the node closest to an event source. RUGGED switches forwarding modes depend on the information available: if no gradient information is available in a region (i.e. far away from phenomena), then flooding is utilized; in the gradient information region, a greedy forwarding approach is utilized whenever distance improvement is possible, or else probabilistic forwarding is used to escape local minima. The parameters of the probabilistic forwarding can be varied depending on the sensor readings.

9.4.4 Trajectory-based forwarding (TBF)

In WSN where nodes in the network all have reasonably accurate location information (either directly through GPS or through the implementation of a network localization technique), a unique approach to efficient querying is the use of pre-programmed paths embedded into the query packet. The geographic trajectory-based forwarding (TBF) technique [148] provides this functionality.

The source encodes a trajectory for the query packet into the header. The trajectory could be anything that can be represented in a parametric form $(x(t), y(t))$ (though non-parametric representations are also possible in principle). For instance a packet to be sent along a sinusoidal curve in a single direction would have the trajectory encoding $(x(t) = t, y(t) = A \sin t)$; and, to travel on a straight line with slope α, it would have the encoding $(x(t) = t \cos \alpha, y(t) = t \sin \alpha)$.

During the course of the forwarding, the ith node that receives the packet with the encoded trajectory determines the corresponding time t_i as the value of t that corresponds to the point of the curve closest to its location (if the curve passes by the same location more than once, then additional information, such as the parameter value chosen by the previous node in the forwarding, may be utilized to determine t_i). This node then examines its neighboring nodes to determine which of them would be most suitable to forward the packet to, depending on $x(t)$, $y(t)$, and t_i. To make progress on the trajectory, the next hop neighbor must have a parameter value t_{i+1} higher than t_i.

The next hop can be determined in many ways depending on design considerations, such as by (a) picking the neighbor offering the maximum distance improvement, (b) picking the neighbor that offers the minimum deviation from the encoded trajectory, (c) picking the node closest to the centroid of the candidate neighbors, and (d) picking the node with maximum energy. Repeating this process at each step, the packet will follow a trajectory close to that specified by the parametric expression in the packet. This is illustrated in Figure 9.4. Good features of this technique are that the trajectory information can often be represented quite compactly, a number of different types of trajectories can be encoded, and the forwarding decisions at each step are local and dynamic. The denser the network, the more accurately will the actual trajectory match the desired trajectory.

While it has many possible applications, TBF is uniquely suited for propagating queries within the network. When a set of possible locations must all be visited, TBF provides an efficient way to guide the query.

9.4.5 Active query forwarding (ACQUIRE)

ACQUIRE [177, 178] is a querying technique involving active query forwarding. The idea is to treat the query as an intelligent entity that moves through the network searching for the desired response. If the query is a complex query, its component sub-queries can be resolved partially en route. As shown in Figure 9.5,

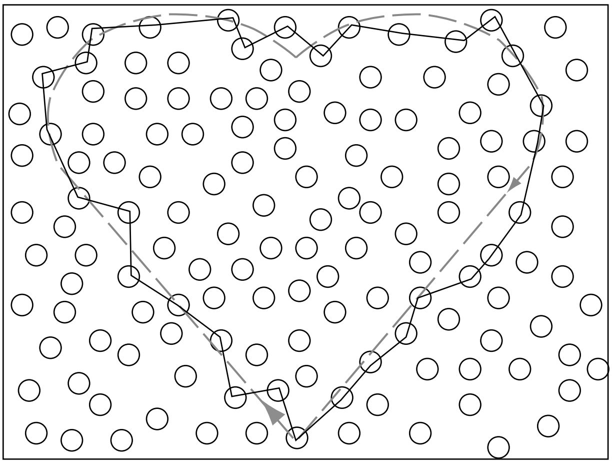


Figure 9.4 Trajectory-based forwarding

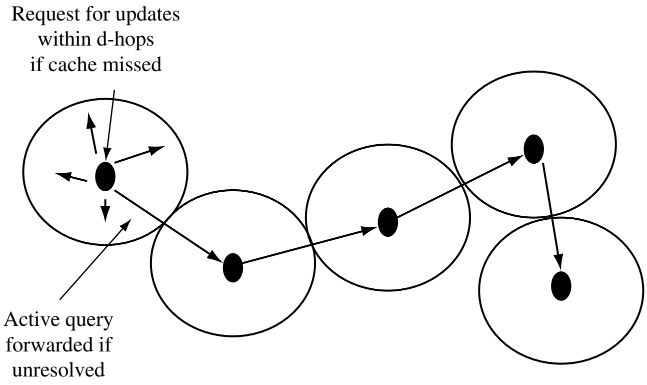


Figure 9.5 ACQUIRE

ACQUIRE progresses through the network as a repeated sequence of three parts:

1. Examine cache: When the query arrives at a node in the network (referred to as an active node), the node first checks its existing cache to see if its cache is valid/fresh enough to directly resolve the query (or parts of it). If the entire query can be resolved from the active node's cache, the response is returned to the sink, else the process continues as follows.
2. Request updates: If the cache does not contain the information desired, the active node issues a request for updates from nodes within a d-hop neighborhood. The responses from the controlled flood are then gathered back and used to see if the query can be resolved.

3. Forward: If it has not already been resolved, the query is then forwarded to another active node (chosen either randomly or through some guided mechanism such as TBF) by a sufficient number of hops so that the controlled flood phases (described below) do not overlap significantly.

A key observation about ACQUIRE is that the look-ahead parameter offers a tunable tradeoff between a trajectory-based query when $d = 0$ (which could be either a random walk or a guided trajectory, depending on the implementation) and a full flood when $d = D$, the diameter of the network. There is a tradeoff for different values of the look-ahead parameter d; when the value of d is small, the query needs to be forwarded more often, but there are fewer update messages at each step. When d is large, fewer forwarding steps are involved, but there are more update messages at each step.

The optimal choice of d in ACQUIRE depends most on sensor data dynamics, which can be captured by the ratio of the rate at which data change in the network to the rate at which queries are generated. When the data dynamics are low, caches remain valid for a long time and therefore the cost of a large d flood can be amortized over several queries; however, when the data dynamics are very high, repeated flooding is required, and hence a small d is favored.

9.4.6 Rumor routing

The rumor routing technique [15] provides an efficient rendezvous mechanism to combine push and pull approaches to obtain the desired information from the network. In rumor routing, sinks desiring information send queries through the network, while sources generating important events send notifications through the network. These are both treated as mobile agents. The event notifications leave a “sticky” trail of state information through the network. Then, a query agent visiting a node where an event notification agent has already passed will find pointer information on the location of the corresponding source. This is shown in Figure 9.6.

Thus, it suffices for the queries to simply intersect with one of the event notification trajectories, rather than have to locate the event node itself. An additional optimization provided is the ability of event notification agents to propagate information about other events based on the state encountered in intermediate nodes, which can reduce the number of unique agents generated for each event.

The trajectory followed by both the events and the queries can be either a random walk (with some loop-prevention built in), or more directed, e.g. straight lines generated using a TBF scheme. It is shown that substantial improvements

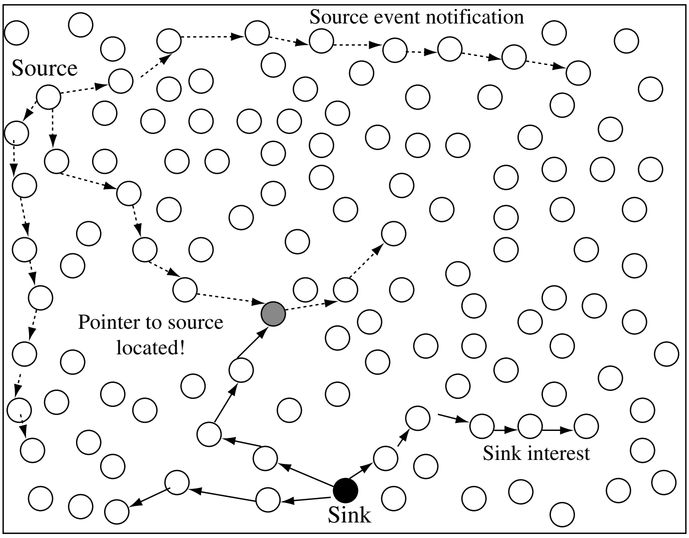


Figure 9.6 Rumor routing

in the energy costs can be obtained by rumor routing compared with the two extremes of query flooding (pull) and event flooding (push).

9.4.7 The comb-needle technique

The same approach as followed by rumor routing, of combining push and pull by looking at intersections of queries and event notifications, is also the basis of the comb-needle technique [124].

In the basic version of this technique, illustrated in Figure 9.7 the queries build a horizontal comb-like routing structure, while events follow a vertical needle-like trajectory to meet the "teeth" of the comb. A key tunable parameter in this construction is spacing between branches of the comb and correspondingly the length of the trajectory traversed by event notifications, which can be adjusted depending on the frequency of queries as well as the frequency of events. To minimize the average total cost per query, the comb inter-spacing as well as the length of the event trajectories should be smaller when the event-to-query ratio is higher (more pull, less push); however, when there the event-to-query ratio is lower, the comb inter-spacing as well as the distance traversed by even notifications should be higher (less pull, more push).

In practice, the frequency of both queries and events is likely to fluctuate over time. An adaptive version of the algorithm [124] handles this scenario. In this adaptive technique, the inter-comb spacing and needle trajectory length are

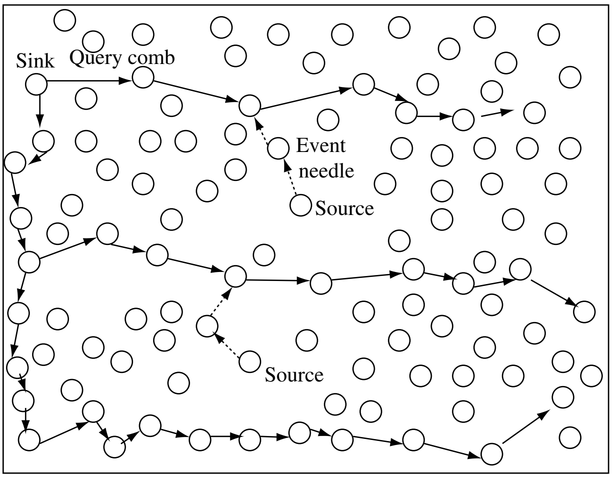


Figure 9.7 The comb–needle approach

calculated and adjusted dynamically in a distributed manner by the sources and sinks by locally estimating data and query frequencies over several observations.

The basic comb structure, with the queries forming the horizontal comb and events the vertical needles is a *global-pull, local-push* model, best suited for conditions when the frequency of queries is less than the frequency of events. When the frequency of queries is more than the frequency of events, and there are multiple querying nodes, then a *reverse-comb structure* can be employed, which would provide a *global-push, local-pull* structure. In the reverse-comb structure, the events form the vertical comb, while queries take horizontal needle trajectories to match with events.

9.4.8 Asymptotics of query strategies

An analytical study of various query strategies by Shakkottai [192] considers three types of queries, all executed using random walks that time out on average after t time units: (i) the source-driven query, in which a random walk from the source attempts to reach within some ϵ region near the location of the desired information, (ii) a replication strategy, where the information is replicated regularly throughout the network, but the search is still source driven, and (iii) a source–receiver "sticky" search, in which, as in the rumor routing and the comb–needle strategies, both the source and destination initiate k random walks and it suffices for any source-initiated walk to intersect with any receiver-initiated walk. The walks are modeled as continuous Brownian motions on the 2D plane. All strategies are measured in terms of the following metric: the decay of the probability that the query is unsuccessful with respect to t, the time duration of the query. Intuitively, the faster this decay rate, the more efficient the query, as a small time duration will suffice to locate the desired information with high probability. It is shown that the simple source-driven search decays as $(\log(t))^{-1}$; with distributed replication, it decays as approximately t^{-1}; while, with the "sticky" search, the decay is given as $t^{\frac{-5k}{8}}$. Thus the sticky search even outperforms distributed replication, so long as the number of push/pull strands is at least 2. Thus, this study provides analytical support for the rumor routing and comb–needle approaches discussed above.

9.5 Data-centric storage and retrieval

Another approach to data-centric networking is to decouple the sensor data storage location from the location where the data are generated. In this structured approach, storage location is carefully chosen based on the type and value of the corresponding data. Instead of blind querying, this enables more efficient retrieval of desired information. It also avoids the overheads associated with pure push-based schemes, where all the data are sent to the sink, regardless of whether they are needed.

9.5.1 Geographic hash tables (GHT)

The use of geographic hash tables [172] provides a simple way to combine data-centric storage with geographic routing. It is quite simple in essence and works as follows. Every unique event or data attribute name that can be queried for is assigned a unique key k, and each data value v is stored jointly with the name of the data as a key value pair (k, v). Two high-level operations provided are Put*(k,v)*, and Get*(k)*. A geographic hash function is used to hash each key to a unique geographic location (x, y coordinate) within the sensor network coverage region. The node in the network whose location is closest to this hashed location (known as the home location for the key), is the intended storage point for the data. When a sensor node generates a new value, the Put operation is invoked, which uses the hash function to determine the corresponding unique location and uses the GPSR geographic routing protocol to route the information to the home node. When the sink(s) issue a Get(k) query, it is sent directly to the same location.

To ensure that the geographic routing consistently finds the same node for a key, and to provide robustness to topology changes, a perimeter refresh protocol is provided in GHT. To provide load balancing in large-scale networks, particularly for high-rate events, GHT also provides a structured replication mode. In this mode, instead of a single location, for each unique key a number of symmetric hierarchical mirror locations are chosen throughout the network. When a node generates data corresponding to the key, it stores it at the closest mirror location, while queries are propagated to all mirror locations in a hierarchical manner

9.5.2 Distributed index for multi-dimensional data (DIM)

DIM [119] is a storage and retrieval mechanism uniquely geared towards multi-dimensional range queries. An example of a multi-dimensional query is "list events such that the temperature value is between 20 and 30 degrees, and light reading is between 100 and 120 units." It comprises two key mappings:

1. All multi-dimensional values are mapped (many-to-one) to a k-bit binary vector.
2. Each of the 2^k possible binary codes is mapped to a unique zone in the network area.

Assume that all values are normalized to be between 0 and 1. The k-bit vector is generated by a simple round-robin technique. If the data are m-dimensional, the first m bits indicate whether the corresponding values are below or above 0.5, the second m bits whether the corresponding values are in the ranges [0–0.25, 0.5–0.75] or in the ranges [0.25–0.5, 0.75–1] (with disambiguation within the ranges provided by the first set of bits), and so on. Consider two examples: let $k = 4$, $m = 2$, the value (0.23, 0.15) is denoted by the binary vector 0000 (which fits all values in the multi-dimensional range (0–0.25, 0–0.25); and the value (0.35, 0.6) is denoted by 0110 (which fits all values in the multi-dimensional range (0.25–0.5, 0.5–0.75).

The mapping of binary codes to zones in a rectangular 2D network area A is performed by the following simple splitting construction: for each successive division, split the region A into two equal-size rectangles, alternating between vertical and horizontal splits. Each division corresponds to a successive bit. If the split-line is vertical, by convention, a "0" codes for the left half, and if the split-line is horizontal, a "0" codes for the top half. This construction, illustrated in Figure 9.8, uniquely identifies a zone with each possible binary vector. In a manner similar to GHT, the node closest to the centroid of the corresponding zone may be regarded as the home node, and treated as the unique point for storage and retrieval.

9.5.3 Distributed index for features (DIFS)

DIFS [67] is a technique suitable for index-based storage and retrieval of information in response to range queries (e.g. "did any sensors report readings of

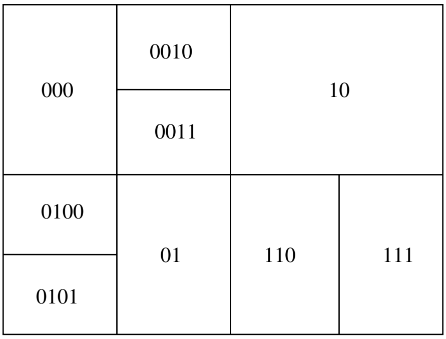


Figure 9.8 Zone creation based on binary index for storing multi-dimensional range data

temperatures within 20–30 degrees?”). DIFS constructs a multiply rooted hierarchical index as follows. Nodes store information for a range of values in a given geographic region. Nodes at low levels cover a wide range of values within a small region, while nodes at the higher levels cover a small range of values within a larger region. In DIFS, each parent has exactly four children, while each child has k parents. Each parent holds information on $1/k$ of the values that a child does, but covers four times its geographic range. A source node measuring an event sends it first to the nearby local index node (determined by a suitable hash function) with a small area coverage and largest range of values. This node then propagates a histogram of observed values to the particular parent at the next higher level with a smaller range of values covering that value, and so on. The leaf index nodes at level 0 point directly to storage nodes, while nodes at level 1 and higher each store four histograms pointing to each of the lower-level index nodes covering smaller areas. DIFS searches may enter at any level of the index structure (and often at multiple points), depending on the spatial extent and value range requested in the query, and drill down to obtain events satisfying the query. The histograms are also useful in resolving more sophisticated queries involving distributions.

9.5.4 DIMENSIONS

A multi-resolution storage and retrieval functionality suitable for spatio-temporally correlated data is provided by the DIMENSIONS architecture [59, 60]. DIMENSIONS incorporates three key components:

1. Multi-resolution hierarchical storage: In DIMENSIONS, the lowest levels of the hierarchy store high-resolution information, while the highest levels store lossy compressed coarse-grained information. Specifically, at the lowest

level of the hierarchy, individual nodes store time series data, possibly with local lossless compression. At each progressively higher level, nodes receive lossy compressed data from multiple children that they uncompress, combine together, and compress to a higher lossy compression ratio, using wavelet compression to send up to the node at the next level. Thus the nodes at higher levels have information about the larger geographic range, but at a coarse grain, while nodes at lower levels have information about smaller regions at a finer granularity.

2. Drill-down querying: Over this hierarchical structure, queries are resolved in a drill-down manner from the top. First responses at the coarsest grain are used to determine which low-level branch is likely to resolve the query, and this process is repeated until the query moves sufficiently down the structure to be resolved.
3. Progressive aging: In practice, such a storage system will face the practical limitation of finite storage. The design principle advocated for such storage in DIMENSIONS is the concept of *graceful aging*. The more extensive fine-grained data at the lowest levels of the earlier hierarchy age and are replaced with incoming new data faster, while the coarse-grained compressed information at higher layers is replaced more slowly. Thus, the farther back in time the data being queried for, the more likely it is that they will be obtained in a summarized form; queries for more recent data are answered with finer granularity.

9.6 The database perspective on sensor networks

So far we have been focusing on a bottom–up networking-centric view of WSN, with an emphasis on networking mechanisms such as routing and query forwarding, albeit enhanced with data-centric notions of in-network processing, data management, storage, and efficient retrieval of queried information. A complementary, top–down perspective is to view sensor networks as a distributed database system, and place the primary emphasis on the interface between the end user (i.e., the human querying the system) and this system. Several researchers have advocated this alternate perspective [13, 129, 233, 66].

9.6.1 Query language

It has been shown that a simple SQL-like declarative language with some extensions can be quite powerful for phrasing a diverse set of queries relevant to sensor networks. The canonical example of such a query approach is the TinySQL/TinyDB implementation of a tiny aggregation service (TAG) [129].

In TinyDB/TAG, all nodes in the network are treated as forming a single table, called sensors. The columns of the table are all the attributes defined for the network and application (e.g. these may be metadata like nodeID, locationID, timeStamp, as well as data readings from the different sensors, such as temperature, light, etc.).

Basic queries in TinySQL can be phrased in the following manner:

```
SELECT {aggregates, attributes} FROM sensors
 [WHERE {selPreds}]
 [GROUP BY {attributes}]
 [HAVING {havingPreds}]
 [EPOCH DURATION {duration}]
```

A simple example is:

```
SELECT max{temperature}, locationID FROM sensors
 WHERE lightIntensity > 120
 EPOCH DURATION 30s
```

The query thus formulated is then flooded through the network and the pertinent information is sent from the nodes that satisfy the relevant predicates. Aggregation operations are applied within the network en route, as the data return to the querying sink (more on this below).

The basic TinySQL language is already quite expressive, enabling users to formulate a wide variety of useful queries. Further enhancements have been proposed [78]. The first is the addition of an event-driven query mechanism, allowing a construct like:

```
ON EVENT fire-detected
SELECT max{temperature}, locationID FROM sensors
...
```

The query is then activated only when the named event occurs. Another enhancement is to allow for storage of buffered data locally within the network. It is demonstrated that, with these slight changes, the expressive power of the language is increased greatly. For example, an illustration given in [78] is a simple high-level program consisting of less than 25 lines that tasks the whole network to track a moving target with query handoffs (i.e. only nodes near the target are activated and provide information on it as it moves). This suggests the possibility of easy high-level programming of sensor networks using an SQL-like language.

9.6.2 Aggregate queries

In TAG, the responses to queries are routed up a tree, with aggregation operators such as MAX, SUM, COUNT etc. applied at each step within the network. Aggregates are implemented via three functions: a merging function f, an initializer i and an evaluator e; e.g., if f is AVERAGE, then given partial state records $PSR1 = < S_1, C_1 >$ and $PSR2 = < S_2, C_2 >$ (S and C for sum and count respectively), $PSR3 = f(PSR1, PSR2) = < S_1 + S_2, C_1 + C_2 >$. The initializer i gives the partial state record for a single sensor reading; e.g., if the single sensor value is x, then $i(x)$ returns $< x, 1 >$. The evaluator e performs the computation on a state record to return the final result; e.g. $e(< S, C >) = S/C$.

The communication savings due to aggregation within the network depend very much on the type of aggregate used. Aggregates such as COUNT, MIN, MAX, SUM, AVERAGE, MEDIAN, HISTOGRAM, etc. all have different behaviors. A classification of these aggregates along multiple dimensions, such as duplicate sensitivity and the size of partial state records, is given in [129] and used to compare aggregation performance.

9.6.3 Other work

There are several other interesting works pertinent to the database perspective on WSN. In GADT [50], a probabilistic abstract data type suitable for describing and aggregating uncertain sensor information is defined. The temporal coherency aware network aggregation (TiNA) technique [194] provides for additional communication optimization by providing temporal aggregation – data values that do not change from the previous value by more than a tolerance level θ are not communicated. Shrivastava *et al.* propose and analyze data aggregation schemes suitable for medians and other more sophisticated aggregates, such as histogram and range queries [197]. The problem of aggregation operators over lossy networks is addressed by Nath *et al.* [144], who provide an analysis of synopsis diffusion techniques that provide robust order and duplicate insensitive aggregation that is decoupled from the underlying multi-path routing structure.

Yao and Gehrke [233] discuss taking into account available metadata about the state of different parts of the network to provide an optimized query plan distributed across query proxies on sensor nodes. The query plan describes both the data flow within the network as well as the computational flow within the node. Bonfils and Bonnet [12] address the problem of optimizing the placement of query operators for long-standing queries autonomously within the network through an exploratory search process.

9.7 Summary

Unlike traditional communication networks that must support a wide range of applications (some not even known at design time), WSNs are much more application specific in nature. Communication in a WSN is most often pertinent to the information available at sensors or desired by an external user. A data-centric approach, where the routing is based on named data rather than addresses, can be advantageous for two reasons: (a) it eliminates the overhead associated with name binding and (b) it allows for energy efficiency through in-network processing, including compression and aggregation of information. The directed diffusion routing mechanism is unique in routing based on named attributes rather than traditional IP-style addressing.

Several studies, including the cluster-based LEACH protocol and many analytical studies, have examined the problem of routing with in-network compression in sensor networks. The studies suggest that, while finding optimal joint routing–compression routes may be difficult, good approximations are possible. It is possible to achieve near-optimal energy performance for routing with compression with a simple LEACH-like clustering technique that is not correlation aware.

Besides end-to-end routing, data discovery and querying form an important communication primitive in sensor networks. Alternatives to the high-overhead naive flooding approach are desired. Several querying techniques have been proposed and analyzed, including expanding ring search, IDSQ, and ACQUIRE. Rumor routing and the comb–needle approach both advocate hybrid push–pull rendezvous techniques, where query trajectories from sinks intersect with event notification trajectories from sources, and show that they can offer significant gains.

Data-centric storage techniques including GHT, DIM, and DIFS offer another alternative by decoupling the location of data storage from the location where data are generated. Data are indexed at locations that depend upon the named content, which makes retrieval much easier with lower overheads than blind querying. The DIMENSIONS technique advocates multi-resolution storage with graceful aging so that more recent fresh information is available at a finer granularity than older data.

Finally, a complementary perspective to data-centric routing and storage in sensor networks is to treat them as distributed databases. TinyDB is a key effort in this direction, advocating the use of a simple-yet-powerful SQL-based declarative query language and in-network processing of aggregate queries.

Exercises

9.1 *Analysis of push versus pull:* Consider a very simple mathematical model to analyze push diffusion and pull diffusion routing. Time is divided into multiple epochs. In each epoch, with probability p_D there is one active source in the network (unknown to the sinks) that generates data, and with probability p_I there is one active sink in the network (unknown to the sources) that is interested in the data. In pull mode, the active sink floods its interest to all n nodes, and, if there is an active source, it responds directly with data along the shortest path to the active sink. In push mode, the active source floods an event notification to all n nodes, and, if there is an active sink, it responds with a reinforcement message down the shortest path to the active source, which can then send data along this path to the active sink. Assume the shortest path between an active source and an active sink is always of length $\sqrt{n}$. Assume interest messages are of size I, event notification messages are of size N, and data messages of size D. Derive an expression for the condition under which push incurs less overhead than pull, and explain it intuitively.

9.2 *Expanding ring search:* Consider an $n \times n$ grid of sensor nodes, where each node communicates with only its four neighbors. Queries are issued for a piece of information located at a random node in the network by a node at the center through a hop-by-hop expanding ring search.

(a) Derive an expression for the expected number of steps until query resolution.

(b) What is the expected number of steps when the information is located at two random nodes (instead of one)? Comment.

9.3 *Trajectory-based forwarding:* How should a circle be represented in parametric $(x(t), y(t))$ form? Simulate the deployment of a 100 node random $G(n, R)$ network with $R = 0.2$ in a unit area. Using any convenient forwarding rule, show the nodes visited by a TBF query that aims to follow the big in-circle centered in the middle of the area with radius 0.5.

9.4 *Rumor routing:* Simulate rumor routing on an arbitrary network of n nodes using random walks as trajectories. Vary the number of source- and sink-initiated walks and quantify the tradeoff between energy cost and latency of query resolution with increasing numbers of walks.

9.5 *DIM binary code mapping:* Give the binary codes that correspond to the following values if $k = 8$: (a) (0.23, 0.15), (b) (0.35, 0.6), (c) (0.83, 0.29).

9.6 *DIM zone creation:* In a square area, draw the regions that correspond to the following codes: (a) 1010, (b) 1101, (c) 00001.