# Deep connections with microcontroller

Presented by developerWorks, your source for great tutorials

**ibm.com/developerWorks**

## Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

# Section 1. Introduction

## Roadmap

The aim of this tutorial is to tie together concepts conveyed in *earlier tutorials* into a cohesive and interesting communications project. This tutorial introduces microcontroller programming topics and techniques useful for a variety of embedded applications. The final product of this tutorial is a prototype navigation subsystem for a wireless robot, affectionately named IrBot. The tutorial includes three significant areas of wireless design: hardware, embedded software, and a remote control application running on a Palm OS device. Full source code and circuit diagrams are presented and discussed.

This tutorial is organized into the following sections:

* Introduction & motivation
* Communications options for embedded devices
* Application design: IrBot
* Building the embedded application
* Creating the PalmOS Project: **`IrBotPilot`**
* Test flight
* Summary

---

## Who should take this tutorial?

The material presented builds upon lessons learned in the introductory tutorials on *Microcontroller application development* and the *Palm OS tutorial series*. The introductory tutorials are considered prerequisites for getting the most out of this tutorial. In particular, the skills and background necessary for getting the most out of this tutorial include:

* C programming experience, with exposure to both Palm OS and serial communications.
* Comfort level with reading schematics and circuit diagrams.
* Familiarity with both serial communications theory and IrDA physical layer protocol. For more information on IrDA, refer to the IrDA links in Resources on page 22 .

---

## Source code map

This tutorial briefly covers the high points of the IrBot's circuit design, with most attention paid to the software portions of the project, including both the IrBot's routines and the Palm OS **`IrBotPilot`**'s code. The source code presented in the tutorial reflects key design considerations and programming techniques for working with embedded devices and infrared communications.

The following functions are found in the **`IrBotPilot`** remote control application

running on a Palm OS device.

* **StartApplication** -- This function initializes the Palm OS application, including the communications resources.
* **ButtonHandling** -- Each button press in the **IrBotPilot** application invokes a specific command. This function is one of five functions responsible for routing the navigation commands accordingly.
* **ProcessCommand** -- This function interprets the requested command and translates it into action via data transmission to IrBot.
* **StopApplication** -- This function runs just prior to the Palm OS application termination, releasing resources.

The following functions are found in the IrBot embedded application running in the microcontroller.

* **Initializeports** -- Upon startup, IrBot takes care of some initialization, preparing for communication with **IrBotPilot**.
* **Feeback** -- This code displays a pattern on IrBot's LEDs. Without a display or other feedback mechanism, it is important to make the most use out of IrBot's few resources.
* **GetCommand** -- This routine receives commands sent by the **IrBotPilot** application.
* **ProcessCommand** -- This code validates received commands and processes them appropriately.
* **Main** -- The main function governs the overall operation of IrBot.

---

# Tools

The following resources are required to complete this tutorial.

* Palm OS application development environment including Palm OS SDK 3.5 or later. For more information on available Palm OS development environments, see *http://www.palmos.com/dev/tools/*.
* Access to a Microchip application development environment. MPLAB is a downloadable software package available from Microchip. In addition to the MPLAB, a MPLAB compatible C compiler is used to construct IrBot's embedded application. See Resources on page 22 for a link to Microchip's Web site.
* Access to IrBot's required bill of materials (available in Part selection on page 10 ).

---

# Some helpful terms

* *Signal Ground* -- Communication signals are often sent as voltages on two or more connected wires. The signal ground pin on a communications device provides a reference signal used to measure voltages.
* *Transceiver* -- A transceiver chip provides both transmitting and receiving data services according to a specific protocol. Transceivers are available for a variety of protocols including serial, Ethernet, Universal Serial Bus, Infrared, Bluetooth and others.
* *Communications Medium* -- A transport for communications signals. Common

examples include copper wire, fiber optic cable, radio frequency spectrum, and infrared light.

* *Infrared* -- A term used to describe light beyond "red" in the spectrum of light frequencies. Infrared is invisible to the human eye and is used in many communications and imaging applications. The most popular infrared application is in the form of a remote control for home consumer goods such as VCRs and televisions.
* *Bluetooth* -- A radio frequency technology aimed at short haul connections and mini networks known as pico-nets.
* *Integrated Circuit* -- Commonly referred to as simply an "IC", the integrated circuit is a set of pre-packaged electronic circuitry with a specific purpose. For example, there are integrated circuits available for data conversion, complex timing sequences, or even voice recording/playback.

---

# About the author

After his college basketball career came to an end without a multiyear contract to play for the L.A. Lakers, Frank Ableson shifted his focus to computer software design. He enjoys solving complex problems, particularly in the areas of communications and hardware interfacing. When not working, he can be found spending time with his wife Nikki and their children. You can reach Frank at *frank@cfgsolutions.com*.

# Section 2. Communications options for embedded devices

## Microcontroller communications

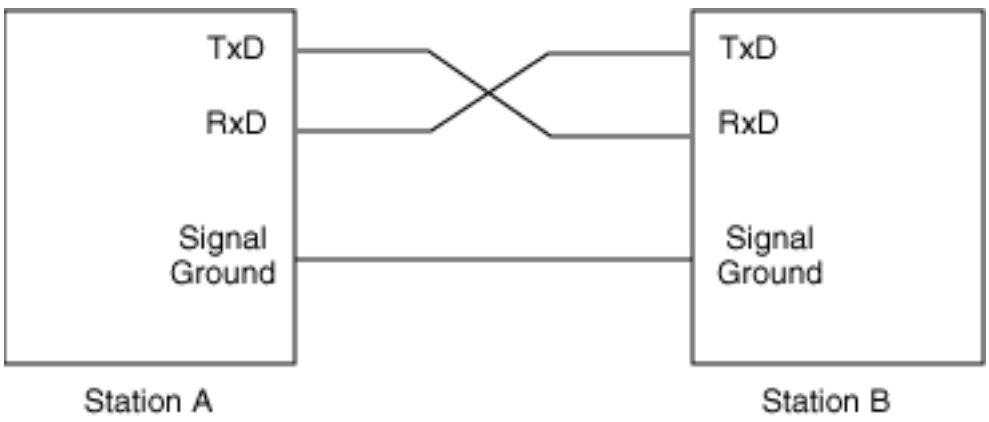
Communications capabilities are an attractive functionality for an embedded device. The ability to communicate affords the device opportunities for interaction with other devices, data collection, data polling, remote access, or even field programming for software updates. This tutorial demonstrates communications between an embedded device and a portable computer, namely a Palm OS powered device. The Palm OS device sends navigation instructions to the embedded device that are subsequently carried out by the embedded device. Before diving into the specifics of the sample application, this section explores some popular communications options for embedded devices.

---

## RS-232 communications, part 1

Most embedded device communications fall into the category of *point-to-point* . In the point-to-point scenario, exactly two devices engage in a "private" conversation." A point-to-point conversation entails a single, logical "connection" between two devices with an organized series of messages going back and forth. By far the most widely deployed of these protocols is known as RS-232 communications. RS-232 is the protocol governing the communications across the "COM port" on virtually every modern computer.

RS-232, also referred to as simply *serial communications,* is often considered the least common denominator in communications options. RS-232 communications requires a simple configuration, consisting of just a few wires between the devices. In fact, communications can be accomplished with as few as two copper wires. This configuration allows one device to communicate with a second device, or vice versa. The transmitting device converts data messages into bits. These bits are combined into a series of voltage pulses across the wires connecting the devices. The diagram below describes common RS-232 signals.




---

# RS-232 communications, part 2

The voltage pulses discussed in RS-232 communications, part 1 on page 5 are measured relative to a *ground* signal shared by both nodes. The receiving system collects these pulses and recombines them into the original message.

When only one device may transmit a message at any given point in time, the link is characterized as *half-duplex* . A traditional, fully-wired RS-232 connection permits data transfer in both directions, simultaneously. When a link communicates in both directions, it is called *full-duplex* . Additional signals provide hardware *handshaking*. Handshaking lines coordinate medium access in half-duplex links as well as assist in providing controls allowing higher data rates between the devices. Common data rates in an RS-232 link range from 110 bits per second to 115,200 bits per second. For more information on the RS-232 standard, refer to the link in Resources on page 22 .

---

# Multinode networks

There are environments and applications where more than two devices must communicate with one another. These generally fall into two categories: a *master/slave* network, or a *peer-to-peer* network. In a master/slave network there is one device controlling the conversation, known as the master device. The master device is responsible for initiating and coordinating all communications in the "network." Typically, the master device sends a message to one of the slave devices by transmitting a message with the target device's "address" as the first portion of the message. All slaves on the network monitor every message, ignoring those with destination addresses other than their own. The slave device with the matching target address is permitted to respond to the request from the master. The query/response mechanism controls access to the data channel and permits many devices to share the same physical "connection."

---

# Peer-to-peer networks

In a peer-to-peer network, each node cooperates with others when communicating across a shared medium or data channel. Coordinating access to the medium in a peer-to-peer network can be a challenging task. The topic of collision detection and collision avoidance comes to the forefront of protocol design. Due to the complexities of collision detection and avoidance associated with peer-to-peer networks, the master/slave approach is much more common for embedded applications.

---

# The RS-485 network protocol

RS-485 is a common multinode network protocol. The RS-485 network connects multiple devices or nodes via the bus architecture. To participate in the network, each device *taps* into a pair of wires representing the RS-485 bus. Data is conveyed along the RS-485 network by a voltage differential between the two wires of the bus. This voltage-differential characteristic, known as a *balanced line*, permits RS-485 networks

to communicate at much higher data rates and over longer distances than an RS-232 link. The data rate is reduced for very long rates; however, data rates of 10 million bits per second over a short distance or a link distance of up to 4000 feet are achievable. Jan Axelson's book, *Serial Port Complete*, provides an excellent description of RS-485 networks. See Resources on page 22 for more information on the book.

## Transceivers, or bit banging

Just how does a microcontroller communicate on a serial data link or network? There are two options generally available to an embedded device. Some microcontroller parts come equipped with special circuitry specifically designed to support serial communications. This circuitry is known as a USART, which stands for Universal Synchronous Asynchronous Receiver Transmitter (see Sidebar: USART on page 7 ). The USART simplifies the action of sending and receiving data on a serial link. Sending and receiving information on the serial data channel is as simple as writing/reading a register or memory location; very little knowledge of the underlying serial protocol is necessary. The downsides are the additional cost associated with the USART and the potential limitation of staying within the bounds of a predefined communications standard. While this may seem odd to worry about, there are scenarios where an application requires non-traditional communications and features a USART does not permit.

In the cases where a USART is either not available or not desired, the alternative is a programming practice known as *bit banging*. Bit banging requires an intimate knowledge of the serial communications protocol. To send information, a particular pin on the microcontroller is set to a high or low state for a period of time. After the appropriate period of time, known as a *bit-time* , has elapsed, the next bit is transmitted. Again, a pin is either raised or lowered to transmit the next bit. Each bit is collected and re-combined into the original message. Obviously, there is more involved in bit banging than in employing a USART, but it is a common and useful approach for adding communications to embedded applications.

## Sidebar: USART

A quick note about the "S" in USART. The microcontroller's USART is capable of transmitting both synchronously and asynchronously. Synchronous transfer is usually meant for "in-circuit" communications, while asynchronous communications are used for interacting between devices. The communications port on Palm OS devices and personal computers are not capable of transmitting synchronously. Those devices have a UART, i.e., no synchronous capability. This tutorial is an example of inter-device communications and thus uses asynchronous communications protocols exclusively.

## Communications chips

Microcontrollers operate in terms of on and off, or 1s and 0s (ones and zeros). When an application intends to send a bit, either via a USART or the bit-bang method, a pin

on the microcontroller transitions from high to low and back repeatedly to transmit that message. However, the communications protocols themselves define voltage levels that do not coincide with the voltage levels employed by the microcontroller. The microcontroller must be insulated from these incompatible voltages; thus it is common practice to employ specialized integrated circuits (IC) in the design. The IC provides voltage conversions and protections to the microcontroller. Additionally, designing around a proven IC affords the designer some confidence that the resulting application complies with the appropriate communications standard. ICs for this purpose are available from Maxim, Linear, Dallas National, and others. For more information, refer to Resources on page 22 . Jan Axelson's book covers this topic very thoroughly. In particular, she discusses some interesting alternatives to using these chips under special circumstances.

Beyond the voltage differences found in various communications protocols, the signaling semantics of the protocols themselves are often best handled by a specialty integrated circuit. For example, there are chipsets dedicated to the RS-232, RS-485, 900 Mhz, infrared and other protocol specifications. The circuit design in this tutorial leverages a specialty chip known as the MCP 2120. The 2120 is used for infrared communication.

---

## With or without wires

Copper wires are not the only option for short-range communications. Other options include infrared and radio frequency links. The Infrared Data Association (IrDA) and the Bluetooth Special Interest Group (BSIG) have specified standards for short-range communications without wires. These technologies are of particular interest and function to embedded applications as they offer "cable replacement" opportunities. Allowing an embedded device to operate without a physical tether permits a reduction in design real estate by eliminating a DB-9 or modular jack and replacing it with either an internal antenna or small infrared LEDs.

Saving physical space may or may not be crucial to a particular device design; however, employing a wireless connection introduces some flexibility for servicing a device or collecting data in the field. Most popular PDAs support these wireless technologies, allowing a new breed of device interactions in the field. For more information on PDA data collection approaches, refer to the *developerWorks* tutorials available on the Palm OS (see Resources on page 22 ).

# Section 3. Application design: IrBot

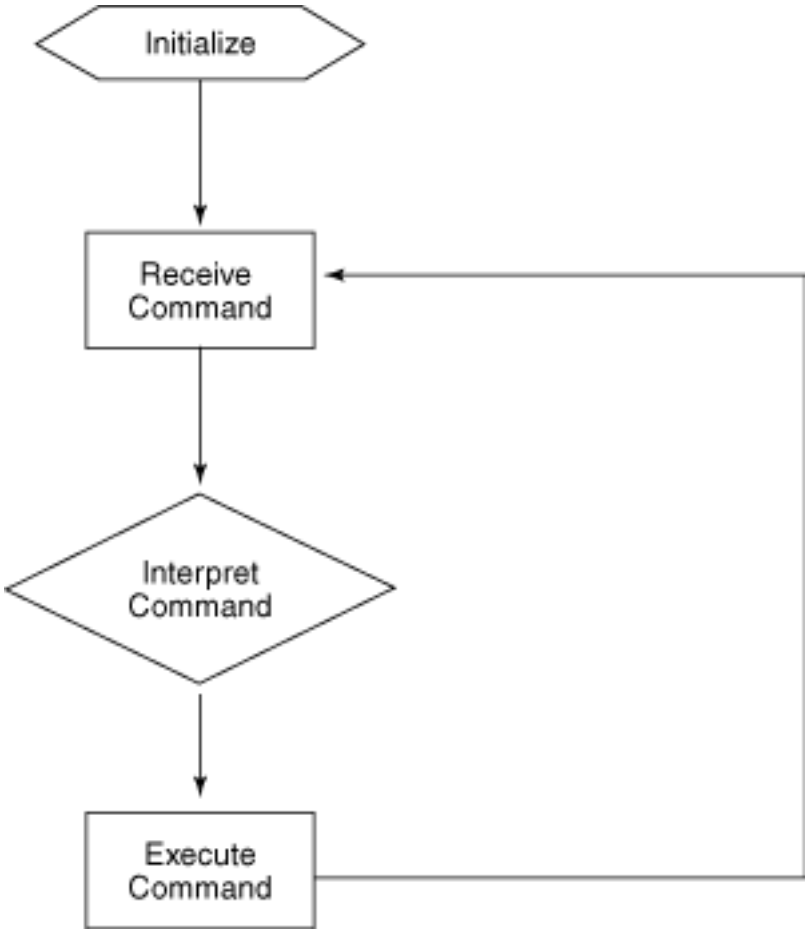
## High level design of navigation system

The focus of this tutorial is a navigation subsystem for a robot or similar device. The tutorial does not cover the mechanical or the power aspects of such a device's locomotion or other features. The sample applications and circuit discussed in this tutorial are concerned with the logical control of the device. The device needs to handle instructions for moving forward, backward, stopping, and turning left or right, although actually getting the device to move is an exercise left for another day. The IrBot acknowledges the received command via illumination of a series of LEDs. IrBot is directed via an application running on a Palm OS powered device. The link between the devices employs infrared communications.

---

## Navigation system flowchart

The image below depicts the states and possible action from each state for the IrBot.




Once IrBot is initialized, it continuously attempts to read a command over the infrared channel. When a command is received, IrBot validates it and carries it out via illumination of the appropriate LEDs. IrBot then returns to retrieving commands via the

infrared channel. Note that the application is strictly unidirectional. IrBot does not send any feedback to the **IrBotPilot** application running on the Palm OS device.

## Part selection

IrBot's circuit is built around a popular Microchip 8-bit microcontroller, model 16F84. This chip is one of the most common Microchip parts and was introduced in the first microcontroller tutorial. The 16F84 is considered a low-end device, and as such, does not contain a USART; therefore a bit-banging approach is required. While this adds some complexity to IrBot's application code, it does keep the hardware costs down and provides an opportunity to explore this concept of bit banging.

Microchip's 2120 chip provides an encoding/decoding service, converting IrDA pulses into signals easily recognized by the 16F84. The 2120 acts as a "middle-man," interpreting signals from both the infrared domain and the traditional serial communications domain. Interestingly, the 2120 is itself a specialized microcontroller.

The infrared transceiver is the TFDS 4500 from Vishay (see Resources on page 22 ). This package receives the infrared pulses from the Palm OS device and sends them along to the 2120 part for decoding.

A bill of materials is presented in the table below:

| Item | Quantity |
|---|---|
| MCHP 16F84 microcontroller | 1 |
| MCHP 2120 IrDA converter | 1 |
| TDFS 4500 IrDA transceiver | 1 |
| 7.3728 crystal | 1 |
| LED bank | 8 |
| Misc. resistors & capacitors, jumper wires | handful |

As an alternative to building the circuit from scratch, a MCP2120 developer's kit is available from Microchip (see Resources on page 22 ). This kit is prewired for the infrared portions and provides jumper blocks for connecting to another circuit containing a 16F84 part.

## Circuit layout

The IrBot's circuit consists of two primary chips. The MCHP 16F84 microcontroller drives LED indicators and interacts with the MCHP 2120 infrared decoder. The 16F84 is responsible for initializing the circuit, interpreting signals from the 2120, and displaying results on the LEDs. There are many connections in this circuit; the few pertinent ones are described here.

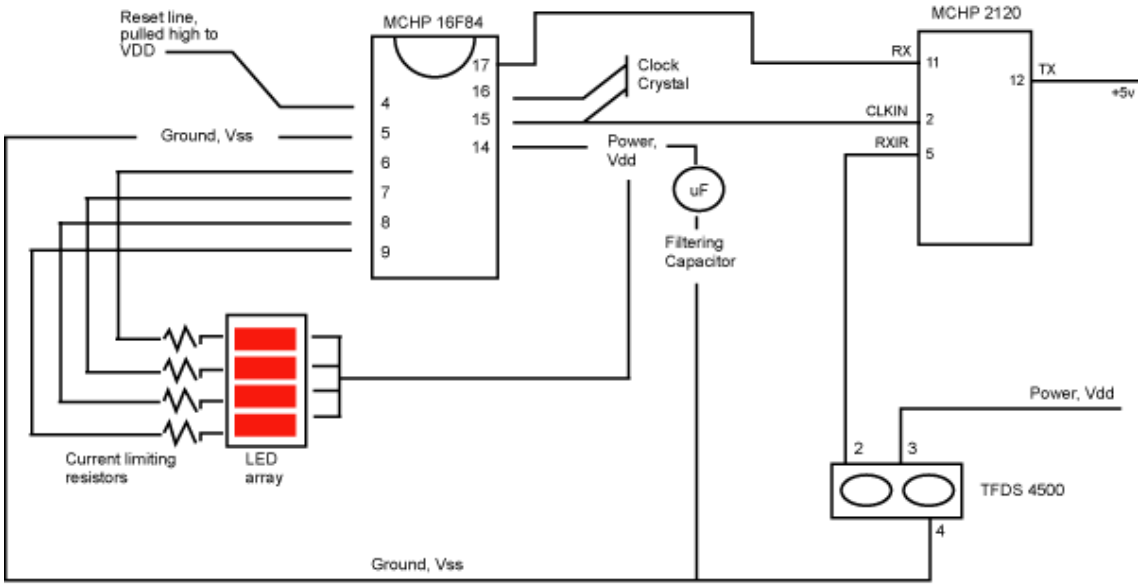
Pin two of the TDFS4500 infrared transceiver is the transceiver's "output." This is connected to pin RXIR (this stands for receive, infrared) on the 2120. Once the 2120 has decoded the pulses from the transceiver, the signal is converted into pulses on the

2120's RX pin. This pin is connected to an input port on the 16F84. Software in the 16F84 interprets the pattern of pulses on this input port, and acts on the data received.

Another key connection in this circuit is the crystal (XTAL) used for timing. Note that a connection exists from pin OSC2 (oscillator, pin 2) on the 16F84 and OSC1/CLKIN on the 2120 part. (CLKIN stands for "clock in".) The two devices share the same clock signal. The XTAL provides timing to the 16F84, which in turn "shares" its clock signal with the 2120.

PORTB on the 16F84 controls the LED bank.

The IrBot application is concerned only with receiving data. As such, the transmit pin on the 2120 part should be "pulled up" to +5v for proper operation. Failure to do so can cause unpredictable results. Please refer to the circuit diagram for more information.




Dropping pins 6,7,8,9 [PORTB.0,PORTB.1,PORTB.2,PORTB.3] low creates a voltage potential, allowing current to conduct through the LEDs, illuminating them.

The next section begins the actual construction of the embedded application software.

# Section 4. Building the embedded application

## Creating the microcontroller project

The MPLAB environment supports multiple tool options. In the introductory Microcontroller tutorial, the sample application was written in assembly language. For this tutorial, IrBot's embedded application is written in the C programming language. C is easier to read and maintain than assembly language. The compiler used for IrBot is PICC lite, a free version of Hi-Tech's C compiler shipped with the PicPlus kit. PICC lite is limited to working with only the 16F84 device and is essentially an evaluation version of Hi-Tech's full featured C compiler. For more information on the Hi-Tech compilers, see Resources on page 22 .

Create a new project in MPLAB using the Project menus. The settings here are similar to the settings used in an assembler project; however, some of the tool settings are different to reflect the installatin path of the C compiler. The best resource for setting up a C project is the tutorial that ships with the PICC lite compiler. See the previous Microcontroller tutorial in Resources on page 22 for more information on setting up a new project in MPLAB.

Here are the pertinent Node Property settings for the C compiler:

* Language Tool: PIC-C Compiler
* Use error file, providing a name in the Data field. For example, **ibmdw_c.error**
* Use map file, providing a name in the Data field. For example, **ibmdw_c.map**
* Use assembler list file.
* Set the include path setting to **c:\picclite\include** (or wherever the PICC compiler's include files are installed)

---

## Initialization of the device

The first activity performed by the IrBot application is the initialization of the I/O ports. The function **initializeports()** does the requisite housekeeping on the ports to enable IrBot to both read incoming data from the MCP 2120 IrDA chip and control the bank of LEDs. Here is the source to the function:

```
void initializeports()
{
   TRISA = 0xff; // set port A to input
   TRISB = 0x00; // set port B to output
}
```

---

## Feedback functions

IrBot has no video monitor or textual output. It has only eight light emitting diodes (LEDs). The state of each LED is controlled through changing the value of the microcontroller's port B. To turn all of the LEDs off, the value 0xff is assigned to the port. The binary equivalent of 0xff is 11111111. This value raises the voltage level to

+5v on each pin, preventing current from flowing through each of the LEDs. To illuminate a particular LED, the appropriate bit in the port must be cleared, or put another way, the bit must be set to zero. Setting a bit to zero has the effect of lowering the voltage on the pin to ground. This creates a voltage potential between the led and the pin allowing electrical current to flow through the LED, thus illuminating it. The **dorollover()** function first turns off all of the leds. Then, one by one, it illuminates an LED, and then turns it back off and illuminates the next LED in the bank of eight.

```
void dorollover()
{
   PORTB = 0xff - 0x00;
   delay();
   PORTB = 0xff - 0x01;
   delay();
   PORTB = 0xff - 0x02;
   delay();
   PORTB = 0xff - 0x04;
   delay();
   PORTB = 0xff - 0x08;
   delay();
   PORTB = 0xff - 0x10;
   delay();
   PORTB = 0xff - 0x20;
   delay();
   PORTB = 0xff - 0x40;
   delay();
   PORTB = 0xff - 0x80;
   delay();
   PORTB = 0xff;
}
```

---

# flash() feedback function

The other useful feedback function is called **flash()**. The flash function takes two parameters. The first is a value representing a bit mask, or a list of LEDs to illuminate. The second parameter tells the function how many times to flash the LEDs. Here is the function:

```
void flash(char x,unsigned int j)
{
unsigned int i;
   for (i=0;i<j;i++)
   {
      PORTB = 0xff;     // turn off leds
      delay();
      PORTB = 0xff - x;
      delay();
   }
   PORTB = 0xff;     // turn off leds
}
```

Here is an example of using the flash function. This line of C code causes IrBot to flash the first and last LED three times:

```
flash(0x81,3);
```

The binary equivalent of 0x81 is: 1000 0001. The binary pattern coincides with the bank of LEDs.

---

# Main program loop

Once initialized, IrBot continuously looks for incoming data from the infrared circuitry. When a command is read, the value is passed to another function for evaluation. When the command has been handled, IrBot resumes listening:

```
void main()
{
   initializeports();
   dorollover();
   while (1)
   {
      processcommand(getcommand());
   }
}
```

---

# Serial communications routine -- GetCommand

IrBot receives data by monitoring the state of a pin on the microcontroller. This PORT A.0 pin is connected to the output of the MCP2120 IrDA chip. Data received by the 2120 chip is converted into signals on its pin number 11. Thus, pin 11 of the 2120 is connected to PORTA.0 on the 16F84 microcontroller. For more information, refer to the circuit diagram above.

The communication protocol between the two chips can be characterized as asynchronous start, and synchronous between bits. This means that the start bit, or the indicator that data is available, can occur at any point in time. However, once this start bit has been received, the subsequent bits are expected within a strict timing sequence. The **`getcommand()`** function continuously waits for the start bit, and then methodically records the remaining bits in each message. If the stop bit is not detected, an error is message is displayed via the flash function. Note that the data is transmitted least significant bit first, or LSB. The following listing contains the code:

```
unsigned char getcommand()
{
unsigned char readbyte;
unsigned char bitcount;
unsigned char mask;
   while (1)
   {
      // look for start bit ....
      if (lookforbit())
      {
         // a bit has been found
         readbyte = 0x00;
         bitcount = 8;
         mask = 1;
         // we should be at the leading edge of the start bit..
                 // let's reposition ourselves
         // to the "middle" of this bit
```

```
         // sleep for a 1/2 bit time,then sample each bit time moving forward
         // byte asynchronous, bit syncrhonous
         delayhalfbitlength();
         if (!lookforbit())
         {
            // we checked to see if we were still on the start bit and were not...
                        false alarm ....
            flash(0xc3,3);
            return 0x00;
         }
         while (bitcount)
         {
            delaybitlength();
            if (lookforbit())
            {
               readbyte = readbyte | mask;
            }
            bitcount--;
            mask = mask * 2;
         }
         // look for stop bit
         if (lookforbit())
         {
            // framing error
            flash(0x81,3);
            return readbyte;
         }
      }
   }
   // should never reach here... put this in to keep compiler from complaining
   return 0x00;
}
```

---

# Robot "actions" -- ProcessCommand

Once a command has been interpreted by IrBot, the **`processcommand()`** function is called upon to carry out the appropriate action. For this tutorial application of IrBot, the command is simply displayed upon the LED bank for a moment and then the LEDs are cycled to indicate IrBot's readiness to receive an additional command.

A real navigation system would interpret the commands to actuate the device in the proper manner. For example, a "left turn" command would result in a stepper motor moving the steering apparatus to the left. Here is the code for simulating the directional commands:

```
void processcommand(unsigned char command)
{
   switch (command)
   {
      case 0x00:
         PORTB = 0xff;
         break;
      case 'A': // stop
         PORTB = 0xff;
         break;
      case 'B': // forward
         PORTB = 0xff - 0x01;
         break;
```

```
        case 'D': // back
           PORTB = 0xff - 0x02;
           break;
        case 'E': // left
           PORTB = 0xff - 0x04;
           break;
        case 'I': // right
           PORTB = 0xff - 0x08;
           break;
        default:
           // display value on LEDs
           PORTB = 0xff - command;
           break;
     }
     delay();
     delay();
     delay();
     delay();
     delay();
     dorollover();
  }
```

## Building and testing the project

Once the code is successfully compiled, load the application into the 16F84 part via the PicPlus (or equivalent) programmer. For more information on flashing the device, refer to the introductory microcontroller tutorial found in Resources on page 22 . When power is applied to the circuit, IrBot should cycle the LEDs. This is an indication that the output portion of the circuit is wired correctly.

The next step is to build the **`IrBotPilot`** application to provide a "remote control" for IrBot.

# Section 5. Building the Palm OS remote controller

## Design considerations

The remote control application running on a Palm OS device is named **IrBotPilot**. This application's sole purpose is communication with the IrBot circuitry. The focus is on function and not form. As such, the application is simply a series of buttons, each indicating a direction or stop. As each button is pressed, a single byte is sent to the embedded IrBot application. To simplify testing, the commands are designated as easily readable characters; this way the application may be tested against another infrared target, such as a PC with an infrared-dongle attachment. Here are the valid commands for **IrBotPilot**:

| Command | Command Data | Description |
|---|---|---|
| A | 0x00 | Stop |
| B | 0x01 | Forward |
| D | 0x02 | Back |
| E | 0x04 | Left |
| I | 0x08 | Right |

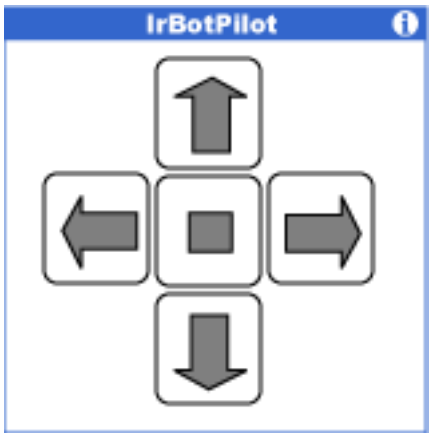
While the **IrBotPilot** application contains only five buttons, note the Command Data for each command. The commands are organized in a *bit mask* fashion. This command set allows for commands to be sent in combination. For example, if the operator desired for the IrBot to move forward and to the left, the Command Data evaluates to 0x05, or F. Of course, some care should be taken to not send contradictory commands such as forward and back at the same time.

---

## Creating the Palm OS project

The **IrBotPilot** project is easy to construct. Begin by creating a new project in Falch.net Developer Studio, or an equivalent Palm OS development environment. See Tools on page 3 for information on obtaining Palm OS development tools. The key ingredient to this application is the ability to interact with a series of Palm OS SDK functions known as the New Serial Manager. For more information on Palm OS development and New Serial Manager, refer to the IBM *developerWorks* series on Palm OS data collection applications (see Resources on page 22 ).

This tutorial's sample, **IrBotPilot**, contains a single form, namely **frmMain**. This form presents each of the navigation controls to the user. Here is a snapshot from the main **IrBotPilot** interface:




## StartApplication

The first function discussed in the sample code is **StartApplication**, which is invoked during the application's initialization. This function opens the communications port on the Palm OS device, sets the data rate to 9600 baud, and switches it into "IrDA" mode. By switching into IrDA mode, the Palm OS device's UART sends IrDA compatible pulses instead of the traditional RS-232 compatible mark and space on the device's physical serial port. For more information regarding the specifics of IrDA Physical layer protocol, refer to the Infrared Data Association for which a link is provided in Resources on page 22 .

```
static int StartApplication(void)
{
   // initialize communications port
   Err e;
   UInt32 flags;
   UInt16 flagssize = sizeof(flags);
   e = SrmOpen(0x8000,9600,&portid);
   if (e)
   {
      FrmCustomAlert(Info,"Unable to Open Communications Port",NULL,NULL);
      return 0;
   }
   // now set the mode to ir enabled
   flags = 0;
   e = SrmControl(portid,srmCtlIrDAEnable,&flags,&lagssize);
   FrmGotoForm(frmMain);
   return 0;
}
```

The variable **portid** is a global variable holding an identifier for the valid communications port. This value is used in subsequent communications actions.

## Button handling

When a button is tapped on the Palm OS device screen, the application handles the event by sending a navigation message to IrBot. Here is the code for the "Move to the Right" button handler and the list of enumerated values available for **IrBotPilot**.

```
static Boolean frmMain_btnRight_OnSelect(EventPtr event)
{
   // Insert code for btnRight
   ProcessCommand(IrBotRight);
   return true;
}
typedef enum
{
   IrBotStop = 0,
   IrBotForward = 1,
   IrBotBack = 2,
   IrBotLeft = 4,
   IrBotRight = 8
}IrBotEnum;
```

Note the bit-mask values of **IrBotEnum**. Using a bit mask affords **IrBotPilot** a future option of allowing multiple, simultaneous navigational choices.

---

# ProcessCommand

The **ProcessCommand** function is responsible for issuing the appropriate value to the IrBot device. Here is the code:

```
static void ProcessCommand(IrBotEnum command)
{
   UInt8 buf[2];
   Err e;
#if TRACENAVIGATION
   switch (command)
   {
      case IrBotStop:
         FrmCustomAlert(Info,"You hit Stop",NULL,NULL);
         break;
      case IrBotForward:
         FrmCustomAlert(Info,"You hit Up",NULL,NULL);
         break;
      case IrBotBack:
         FrmCustomAlert(Info,"You hit Down",NULL,NULL);
         break;
      case IrBotLeft:
         FrmCustomAlert(Info,"You hit Left",NULL,NULL);
         break;
      case IrBotRight:
         FrmCustomAlert(Info,"You hit Right",NULL,NULL);
         break;
   }
#endif
   buf[0] = 'A' + command;
   SrmSend(portid,buf,1,&e);
}
```

Note that this function has a C preprocessor directive (#ifTRACENAVIGATION). If this value is set to 1 during compilation, the user receives an informative message during use of **IrBotPilot**. This is convenient for testing but quickly becomes annoying once all of the buttons' functionality is confirmed.

Also, observe the technique of adding the command value to the letter 'A'. This allows

testing of the **IrBotPilot** application against a simple terminal package. It is easier to read an uppercase letter than a hex value such as 02. In fact, some terminal programs do not support the display of such "non-printable" characters. The IrBot software essentially "subtracts" the 'A' from the message, leaving it with just the desired command.

---

## StopApplication

Every resource should be released upon termination of the program. One of the last tasks handled by **IrBotPilot** is closing down the communications port.

```
static void StopApplication(void)
{
   SrmClose(portid);
   FrmCloseAllForms();
}
```

---

## Building and testing the remote control application

For the first round of testing, it is recommended that the instructions for enabling the infrared functionality be commented out of the code. By doing so, **IrBotPilot**'s output goes directly to the normal serial port of the Palm OS device. Compile the **IrBotPilot** application and install it on the Palm OS device. It is now ready for testing.

Place the device in its sync cradle and run a serial terminal package on the desktop. Be sure that the sync software is disabled to avoid a "port in use" conflict on the desktop. Start **IrBotPilot** and press any of the navigation buttons. Commands should appear on the terminal program's window. In addition, if the TRACENAVIGATION definition is set in the source file (**frmMain.c** in the sample code), informational alerts display on each button press.

If text does not appear on the terminal window, double check that port speeds match and that all cables are properly connected.

Once the application is running satisfactorily, be sure to restore the commands in **StartApplication** to enable the IrDA mode.

The next section brings IrBot and the **IrBotPilot** application together for testing and troubleshooting as necessary.

# Section 6. Test flight

## Controller and device integration testing

Now that both IrBot and **IrBotPilot** are completed and tested on their own, it is time to perform the integration testing. Power up IrBot and initiate **IrBotPilot** on the Palm OS device. If IrBot is operating properly, it responds to each button press with the appropriate LED illuminating. IrBot won't actually move, but it should at least acknowledge the navigation request. Keep in mind that use of the communications port imposes a relatively significant drain on Palm OS device power resources, so don't take IrBot on too long a walk without some extra batteries available.

---

## Troubleshooting

If IrBot fails to respond, check the following:

* Is there power to the IrBot circuit?
* Did IrBot perform the startup routine (cycling thru the LEDs)?
* Did IrBot respond at all, or just not properly?

Double check that the commands issued by **IrBotPilot** are correct by using a desktop terminal program and infrared serial dongle if available. If the commands received by IrBot and illuminated in the LED bank are not the expected values, the timing sequence of the bit banging delay routines may need adjustment. This is particularly true with the use of different crystal speed or baud rate.

# Section 7. Summary

## Summary

This tutorial examined wireless microcontroller application design by reaching beyond traditional wireless discussions of Internet access and efficient handheld user interface design. IrBot and its companion, **`IrBotPilot`**, provide an interesting and functional foundation upon which to build other embedded devices and portable applications. One thing to try with IrBot is to upgrade it to BtBot, by Bluetooth-enabling it. A simple change to the **`IrBotPilot`** software running on the latest model Palm OS devices makes it Bluetooth-enabled. The Resources section provides a wealth of further information on embedded design topics, including Bluetooth and infrared.

---

## Resources

* Sample source code is available at *http://www.cfgsolutions.com*. Follow the links to the left for the IBM *developerWorks* sample code page.
* Jan Axelson's *Serial Port Complete* provides an excellent resource on RS-232 and RS-485 networks. Information on how to obtain this book is available at: *http://www.lvr.com/*.
* The *Infrared Data Association* has a wealth of information about the latest IrDA protocols.
* For more information on Bluetooth, visit *http://www.bluetooth.com* and/or *http://www.bluetooth.org*.
* Maxim manufactures a series of communications integrated circuits. They can be found at *http://www.maxim-ic.com* .
* The first microcontroller tutorial can be found at: *http://www-105.ibm.com/developerworks/education.nsf/wireless-onlinecourse-bytitle/78D64E10D25E6*
* Previous Palm OS tutorials (IBM) are available at: *http://www-106.ibm.com/developerworks/wireless/library/wi-edpick.html*
* Learn more about Palm's Bluetooth offering at *http://www.palmos.com/dev/tech/bluetooth*.
* Microchip is the maker of the integrated circuits used in the IrBot project. Their Web site can be found at *http://www.microchip.com*. Microchip's development tool, MPLAB, can be found at *http://www.microchip.com/1000/pline/tools/picmicro/devenv/mplabi/index.htm*.
* Darrick Addison's *developerWorks* article on Robotics provides some insight into the use of microcontrollers in automation: *http://www-106.ibm.com/developerworks/library/l-rob.html* .
* Aegis Technology's Steve Schlanger provided essential assistance on the hardware design of IrBot. Steve's Web site can be found at *http://www.aegisweb.net*.
* Myke Predko's book, *Programming and Customizing PICmicro Microcontrollers* (McGraw Hill), provides a wealth of information on programming PIC devices.
* Roger Stevens' book, *Serial Pic'n: PIC Microcontroller Serial Communications* (Square 1 Electronics), goes in depth on the topic of bit-banging communications.
* The most up to date list of Palm OS development tools can be found at *http://www.palmos.com/dev/tools/*

---

# Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

---

## Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.