Mobile Phone Controlled PODdevices™ A Tutorial Part 4 - Coding the Led Controller PODdevice™

by Matthew Ford 22nd April 2011

Coding the Led Controller PODdevice™ (Protocol for Operations Discovery)

Outline of the Tutorial

Part 1 – Introduction to PODdevices

- Introduction - Why PODdevices?
- What can PODdevices do.
- Protocol for Operations Discovery (POD).

Part 2 – POD Mobile Phone Application

- PODapp - Choosing your Mobile Phone
- PODapp - Setting up the Connections
- Connecting to your PC
- Installing the PC Terminal Emulator
- Connecting PODapp to TeraTerm

Part 3 – Led Controller PODdevice Messages

- The PODdevice Main Menu
- The On/Off Command
- The Fade On / Fade Off Command
- The Up and Down Commands
- The Set Level Sub-Menu
- Low, Medium, High and Off Commands

Part 1 of the tutorial introduced PODdevices and the covered the specification of the operation discovery protocol. Part 2 of this tutorial covered the mobile phone PODapp. Part 3 covered the design and testing of the PODdevice messages that the Led Controller would support. This part of the tutorial covers the code changes need to turn the RemoteLedDriver into a PODdevice. As a PODdevice, the Led Controller will advertise its functionality to the PODapp and let the user control the Led. For the Led Controller this functionality will include fading the led on and fading it off, increasing and decreasing the brightness and two sub-menus, one to set specific levels, High, Med, Low and another to display the current level setting and led current.

Coding the Led Controller PODdevice™

The complete code for the Led Controller PODdevice can be downloaded from PODLedDriver.asm That code is based on the previously described RemoteLedDriver.asm code. This part of the tutorial will only cover the differences. Please refer to the previous tutorials, Mobile Phone Controlled Led Driver and Bluetooth Controlled Led Driver for descriptions of the other parts of the code.

The PODdevice™ Message Handler

The main difference between the RemoteLedDriver and the PODLedDriver is in the way the incoming serial data is handled. Based on the message design done in Part 3 of this tutorial, the PODLedDriver needs to be able to parse and handle each of the following messages
{.} {o} {f} {u} {d} {l} {a} {0} {1} {2} {3} {i} {v} {m}

This is done in three stages
1) process the received characters looking for the starting **{** character, the command character and the terminating **}** - PROCESS_RECEIVED_CHAR
2) execute the command - PROCESS_CMD_CHAR
3) send back a response message to the PODapp so that the PODapp knows the command has been received and executed. - SEND_...

PROCESS_RECEIVED_CHAR

```
PROCESS_RECEIVED_CHAR:
   cpi Temp, '{' //
   breq PROCESS_RECEIVED_START_CMD
```

```
   cpi Temp, '}' //
   breq PROCESS_RECEIVED_END_CMD

   // else save last char
   tst commandCharacter
   brne RET_PROCESS_RECEIVED_CHAR
   // else save the first
   mov commandCharacter,Temp;
   rjmp RET_PROCESS_RECEIVED_CHAR

PROCESS_RECEIVED_START_CMD:
   clr commandCharacter // clear on msg start
   rjmp RET_PROCESS_RECEIVED_CHAR

PROCESS_RECEIVED_END_CMD:
   mov Temp, commandCharacter
   rcall PROCESS_CMD_CHAR  // process the cmd
   ldi commandCharacter, -1 // finished processing

RET_PROCESS_RECEIVED_CHAR:
ret
```

This routine is called for each character received on the RS232 Rx pin (see Coding the RS232 Module for details). The received character is in the Temp register. When the message start character **{** is received the commandCharacter is cleared. Then then next character is saved as the commandCharacter. Any further character before the closing **}** are ignored. When the message termination character **}** is received then the saved commandCharacter is loaded into the Temp register and PROCESS_CMD_CHAR is called to execute the command. Finally the commandCharacter is set to -1 to prevent any character outside the message start **{** and end **}** characters being saved as a command character.

PROCESS_CMD_CHAR

The PROCESS_CMD_CHAR routine starts with a long switch statement which jumps to the appropriate label to handle that command. If the command is not recognized then the Led Controller just sends back an empty command { } to let the PODapp to let it know it is connected. Since the PODapp should only ever send commands that the Led Controller as advertised, this should not normally happen.

```
PROCESS_CMD_CHAR:
   cpi Temp, 'o' //  turn on/off
   breq PROCESS_RECEIVED_ON_OFF

   cpi Temp, 'f' //  fade on/off
   breq PROCESS_RECEIVED_FADE_ON_OFF
...
other commands here ...
   cpi Temp, 'i' //
   breq PROCESS_RECEIVED_CHAR_SEND_ADC_READING

   cpi Temp, 'v' //
   breq PROCESS_RECEIVED_CHAR_SEND_LIGHT_LEVEL

   // else invalid char so just ignore
   // send back {}
   rjmp PROCESS_RECEIVED_CHAR_SEND_EMPTY_CMD
```

There are three types of commands. Those that just send back data, either menus or the streaming raw data message **{=}** plus data (i e. **{.} {l} {a} {i} {v}, {m}**), those that just set a level (i e. **{u} {d} {0} {1} {2} {3}**) and those that toggle depending on the current state (i .e. **{o} {f}**)

Menu Commands

The commands that send back data or menus just set the appropriate flags to trigger the respective SEND_ routine to send the menu or data. For example

```
PROCESS_RECEIVED_MAIN_MENU:
    // set flag to send menu
```

```
    sbr CMDS, (1<<CMDS_MainMenu)
    rjmp END_PROCESS_CMD_CHAR
```

Then in the MAIN_LOOP, when the SEND_MAIN_MENU routine is called, the main menu response is sent back to the PODapp. There are three main menu messages defined one for when the led is off or fading down and one for when it is on but is not at it maximum level and one for when the led is at its maximum setting.. The BRANCH_IF_TORCH_OFF_OR_FADING_DOWN and BRANCH_IF_TORCH_NOT_MAX macros is used to choose the appropriate response menu to send back.

MAIN_LOOP

The MAIN_LOOP is

```
MAIN_LOOP:
    mov New_TORCH_State, TORCH_State // set initial state
    rcall PROCESS_RS232  // process char if there is one
    // and process cmd if } received
    rcall LED_CONTROL_PROCESSING // each time ADC completes and save value
    rcall UPDATE_TORCH_STATE  // transfer new state to TORCH_State and update setpoint etc.
    //
    // now call all the message routines
    // they will just return if nothing set to send
    rcall SEND_EMPTY_CMD  // send this first if set
    rcall SEND_MAIN_MENU
    rcall SEND_ONOFF_UPDATE
    rcall SEND_SET_MENU
    rcall SEND_DATA_MENU
    rcall SEND_SUB_MENU
    rcall SEND_STREAMING_CMD
    rcall SEND_LEVEL // send level
    rcall SEND_LIGHT // then % light
    rcall SEND_ADC_READING // send Amps if needed
    rjmp MAIN_LOOP
```

As in the Bluetooth Controlled Led Driver , this MAIN_LOOP just runs continually processing characters received from the RS232 input, controlling the Led each time there is a new current measurement and acting on the trigger flags set by the push button debounce code and the PROCESS_RS232 routine. Each SEND_... routine is controlled by its own flag. If the flag is set then that message is sent back to the PODapp. The PROCESS_CMD_CHAR and its support routines set the appropriate flag depending on the command received and the torch state.

BRANCH_IF_TORCH_OFF_OR_FADING_DOWN macro

In a number of places in the code we need to check if the led is currently on or off so that the correct toggle action is taken or the correct menu update is sent. We define the led as being off if either it is off or it is fading down. If it is fading down it will be off very soon. The BRANCH_IF_TORCH_OFF_OR_FADING_DOWN macro is similar to other BRANCH_ macro described in earlier tutorials and will branch to the label argument if the torch is either off or fading down. Otherwise the program flow continues to the next statement.

Level Setting Commands

The commands that just set a level (i e. **{0} {1} {2} {3}**) just set that level e.g.

```
PROCESS_RECEIVED_CHAR_TORCH_HIGH: // 3 == high
   ldi New_TORCH_State,Torch_State_HIGH
   rjmp PROCESS_RECEIVED_CHAR_UPDATE
```

The up and down commands (i e. **{u} {d}**) set the torch state to variable and then increment or decrement the level e.g.

```
PROCESS_RECEIVED_CHAR_TORCH_UP:
   ldi New_TORCH_State, Torch_State_VARIABLE
   // clear levels
   rcall SHIFT_DOWN_LEVEL
   inc Level
   rcall SHIFT_UP_LEVEL
   sbr CMDS, (1<<CMDS_OnOffUpdate)
   rjmp END_PROCESS_CMD_CHAR
```

The call to SHIFT_DOWN_LEVEL and SHIFT_UP_LEVEL are there convert between the 10 user selectable levels and the 40 levels used for fad on and fad off. (e.g.)

```
//-----------------------------------
// Shift down level
// When get u or d cmd want to move level by 4
// so there is only 10 steps between off and MaxLevel 40
// this routine divides by 4 before the level is
// inc or dec
// ----------------------------------
SHIFT_DOWN_LEVEL:
    push Temp
    ldi Temp, Level_Shift
    tst Temp
    breq END_SHIFT_DOWN_LEVEL // nothing to do
SHIFT_DOWN_LEVEL_LOOP:
    lsr Level
    dec Temp
    tst Temp
    brne SHIFT_DOWN_LEVEL_LOOP
END_SHIFT_DOWN_LEVEL:
    pop Temp
ret
```

The SHIFT_DOWN_LEVEL routine divides the Level register by 4 before it is incremented or decremented. The similar SHIFT_UP_LEVEL then multiplies the Level register by 4 again to scale the Level register back into the range 0 to maxLevel (40) These 40 levels are then mapped to an approximately logarithmic scale of current settings, via the Log_SetPoint table, to more closely match the eye's perception of brightness.

```
Log_SetPoint:
// this maps levels into current setpoints between 0 and 1000 (max ADC count is 1024)
     // 0 == 0, 1==16    2==24   3 == 40
     // 0  1  2  3  4  5  6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
.dw     0, 1, 1, 2, 2, 3, 3,  5,  6,  7,  8,  10, 12, 15, 18, 21, 26, 32, 38, 47, 57
    //  21   22   23   24   25   26  27   28   29   30   31   32   33   34   35   36  37   38    39  40
.dw     70,  86, 105, 129, 156, 188, 226, 273, 319, 373, 436, 509, 569, 636, 710,794, 841, 891, 944,1000
```

When the CMDS_OnOffUpdate flag is handled by SEND_ONOFF_UPDATE routine called from the MAIN_LOOP, the SEND_ONOFF_UPDATE routine checks the current level and state of the torch and chooses the correct update to send to the navigation menu.

On/Off and FadeOn/FadeOff Commands

The On/Off and FadeOn/FadeOff commands (**{o} {f}**) are toggles. That is each time the command is received by the PODdevice, the PODdevice toggles the state from on to off or from fadeOn to fadeOff.

The on/off processing looks checks the current Led state using the BRANCH_IF_TORCH_OFF_OR_FADING_DOWN macro, and then sets the Led either off or to Low. The FadeOn/FadeOff processing is similar except that it set the torch state to either Torch_State_FADE_UP or Torch_State_FADE_DOWN.

In the background the TIMER_FADE routine is called every 2mS using the same timer the switch debounce uses. Each time it is called if the FADE_COUNTER has reached zero and torch is in a fadeOn or fadeOff state, the TRIGGER_FADE flag is set and the FADE_COUNTER is reloaded. If fading up, 6 counts are used. If fading down 12 counts are used so that the fade down is twice as slow as fading up.

```
//-------------------------
// TIMER_FADE:
// Called from TIMER1_2mS each 2 mSec
```

```
// FadeUp counts to 6
// FadeDown counts to 12
// 40 steps => 0.5 sec up and 1sec fade down
//-------------------------
TIMER_FADE:
    lds Temp, FADE_COUNTER
    dec Temp
    brpl END_TIMER_FADE
    // else == 0
    ldi Temp, 1 // for next dec unless updated below
    cpi Torch_State, Torch_State_FADE
    brlo END_TIMER_FADE // not fade just load 1 in counter

    // else set trigger
    sbr TRIGGER_Flags, (1<<TRIGGER_FADE)
    ldi Temp,Fade_Count_Up  // reload
    cpi Torch_State, Torch_State_FADE_UP
    breq END_TIMER_FADE  // load up timer

    // else down
    ldi Temp,Fade_Count_Down
    //rjmp  END_TIMER_FADE // drop through

END_TIMER_FADE:
    sts FADE_COUNTER, Temp
ret
```

The TRIGGER_FADE flag is examined in the PROCESS_SWITCH_STATE_TRIGGER routine which is called from the MAIN_LOOP via the LED_CONTROL_PROCESSING routine. If the TRIGGER_FADE flag is set then the FADE routine is called.

```
//--------------------------------------
// FADE
// called every time FADE_TRIGGER is set
// if Torch_State_FADE_UP increase level
// if Torch_State_FADE_DOWN decrease level
// if not FADE do nothing
//
//--------------------------------------
FADE:
   push Temp
   cpi Torch_State, Torch_State_FADE
   brmi END_FADE // not fading so return

   cpi Torch_State, Torch_State_FADE_UP
   breq FADE_FADE_UP

   // else fade down
FADE_FADE_DOWN:
   dec Level
   rcall LOAD_SETPOINT_FROM_LEVEL
   rjmp END_FADE

FADE_FADE_UP:
   inc Level
   rcall LOAD_SETPOINT_FROM_LEVEL
  // rjmp END_FADE fall through

END_FADE:
   pop Temp
ret
```

The FADE routine checks the torch state and increases or decreases the level as appropriate. The LOAD_SETPOINT_FROM_LEVEL routine cleans up levels that are <0 or > maxLevel.

Conclusion

This completes a full implementation of POD (Protocol for Operations Discovery). Under the POD protocol, the PODdevice™ advertises its capabilities to the PODapp™ so the same PODapp can be used to control any PODdevice. The POD protocol is very simple with

minimal message structures and is readily implemented by small micro-processors such as the Atmel ATTINY series.

The first section presented the definition of the POD protocol. The second section presented a functioning implementation of a PODapp for a Java enabled mobile phone. That section also covered selecting a suitable phone and installing the PODapp. This application is universal to all PODdevices and allows multiple devices with different capabilities to be controlled from this one application. The third section covered the design of the PODdevice commands for a sample LED driver. The last section built on the previous Bluetooth Controlled Led Drive and modified the code to make the LED controller a PODdevice.

Contact Forward Computing and Control by email