IR REMOTE

IR REMOTE -- Infrared Remote Control

Murdock Taylor
Design Consultant
EPDDCS
P.O. Box 4739
Cary, NC 27519
USA
(919)460-0081 Voice/FAX
email: murdock.taylor@EPDDCS.com
www.EPDDCS.com

IR REMOTE is an IR Pulse Code Modulation (PCM) Remote Control built around the TI ultra low-power MSP430 microcontroller, Vishay TSOP1833SS3V 33.0 kHz Photo Module for PCM Remote Control Systems, Vishay TSAL4400 Infrared Emitter, 3V Li coin cell battery (CR2032), and the 1 to 5-button key fob style Polycase TB-20 Enclosures. This design will allow professional, low-cost, small, battery-powered, and easily customized IR remote control capability to be added to a project.

There are many applications where a small and relatively simple remote control might be used: light dimmers, overhead fan controls, automated blinds/awnings, heating/air conditioning controls, data acquisition control, instrumentation control, and a wide variety of difficult to access industrial sensors/controls, etc.

IR REMOTE is targeted at any application where ON/OFF, UP, DOWN, MODE, BRIGHTER, DIMMER, FASTER, SLOWER, etc., type of IR remote control commands would make sense, plus where a remote control package needs to have a professional and non-prototype appearance.

The Polycase TB-20 enclosures and their elastomer keypads are inexpensive even in small numbers ($2.30/ea qty 1-9) and can be customized (screen printed, painted, and/or labeled) for far less than having custom membrane keypads/overlays made, or having custom elastomer keypads made, or having custom injection molded plastic cases made.

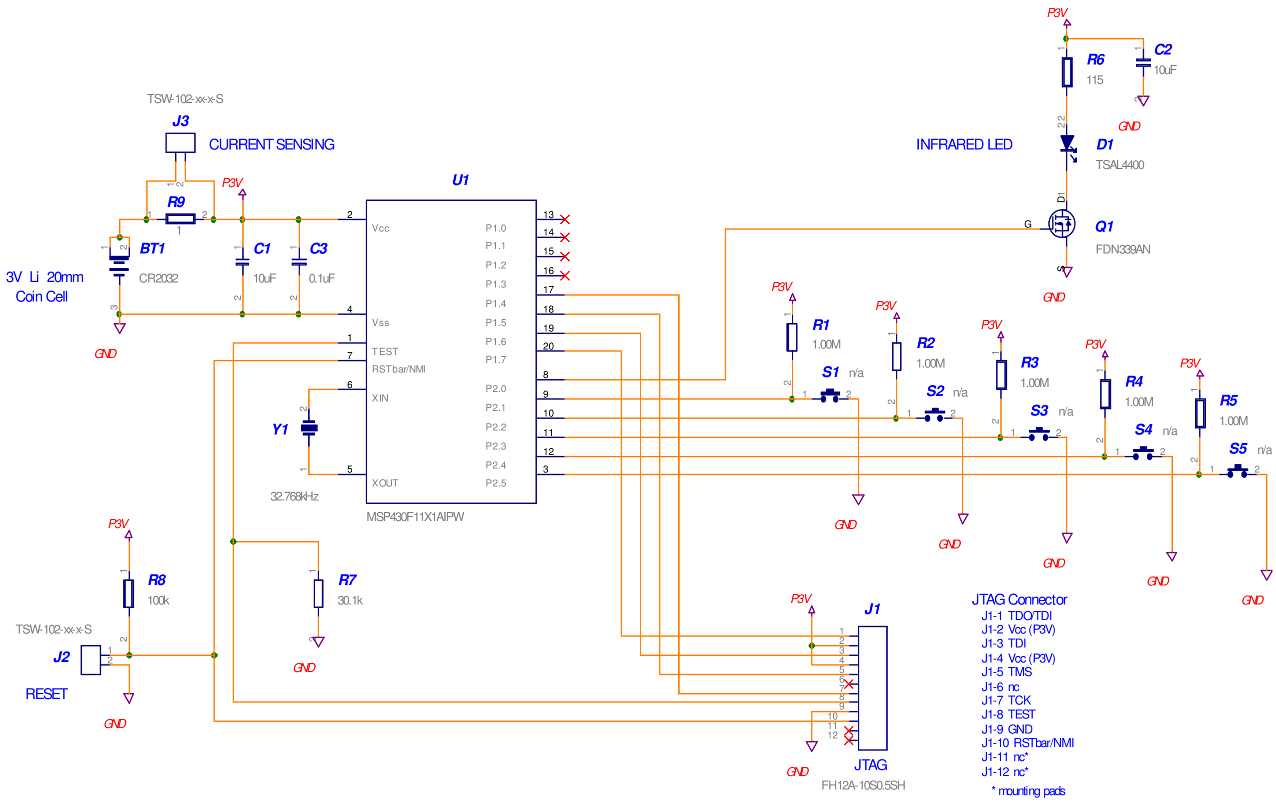
Polycase FB-20 Enclosures (1 to 5 Button Versions)

The IR REMOTE control is a TI MSP430F11xIPW or MSP430F11x1AIPW based design powered by a 3V Li coin cell (CR2032). There are five elastomer button keypad targets on the PCB that match all of the possible 1 to 5 button keypad versions of the Polycase TB-20 enclosures. All five of the keypad button inputs are pulled up to 3V via 1M resistors (see IVEX WinDraft schematic file ➔ ir_remote_a.sch). Pressing an elastomer button will cause the keypad button input to be pulled down to ground.

The keypad button inputs are:

S1 on P2.1
S2 on P2.2
S3 on P2.3
S4 on P2.4
S5 on P2.5.

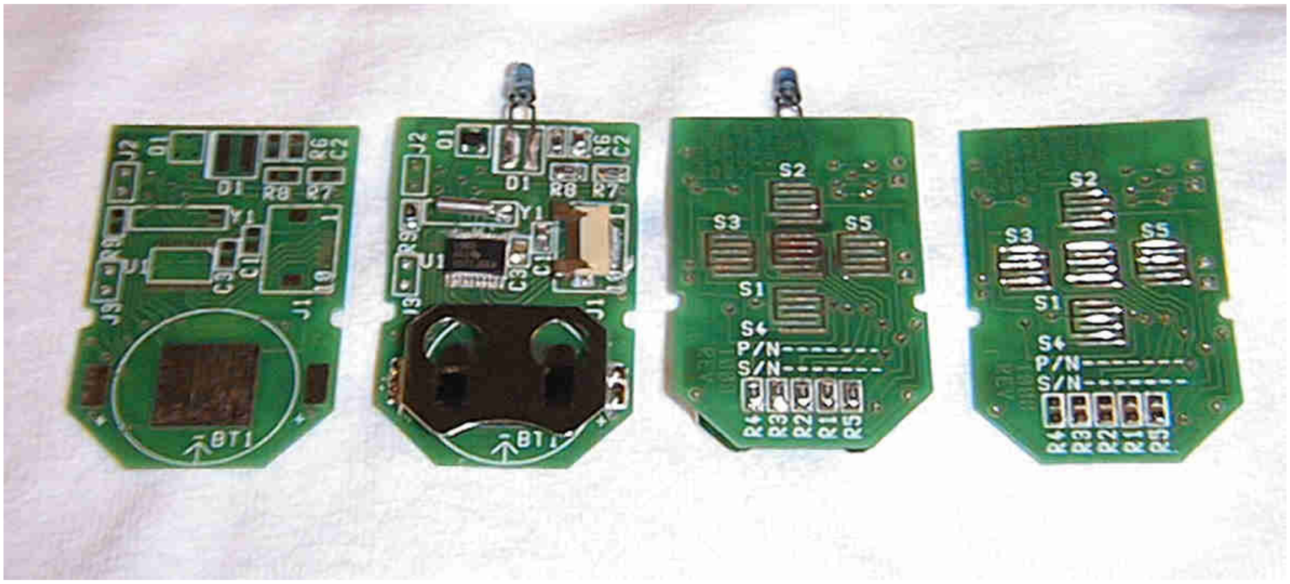
The only output is P2.0. P2.0 drives an N-channel MOSFET that in turn drives the 3mm Infrared LED (Vishay TSAL4400). The IR LED is driven at 32.768kHz, which is conveniently very close to the center of the IR receiver's 33.0 kHz (+/- 5%) carrier frequency. Since the MSP430 has an external 32.768 kHz low frequency crystal, the IR LED modulation can easily be generated from the 32.768 kHz ACLK. The IR LED will draw just under 15mA when ON (this is a design limitation imposed by the pulsed duty limits of the CR2032 coin cell battery). The IR REMOTE is designed to work with a Vishay TSOP1833SS3V 33.0 kHz PCM 3-pin (power, ground, and output) receiver chip incorporated into the design of the product being controlled. The output of the TSOP1833SS3V is a digital serial signal matching the infrared PCM input, but with the carrier removed (active low logic signal = carrier, high = no carrier). The TSOP1833SS3V is a sophisticated part with considerable IR noise filtering and immunity, resulting in robust and reliable PCM reception. Due to the pulse duty limitation of the 3V Li coin cell battery, the IR REMOTE will only work line of sight and will need to be pointed in the direction of the receiver. (Your TV remote control pulses its IR LED at a much higher current due to its alkaline battery(s) and is able to saturate a room with its IR signal so it does not need direct line of sight to work with the TV.)



IR REMOTE Circuit

There are two unpopulated 2-pin headers on 0.100" spacing, J1 and J2. J1 is across a 1 ohm resistor in series with the 3V Li coin cell battery's + terminal. J1 allows the direct monitoring of the current out of the Li battery. J2 is the reset header and shorting its pins produces a reset.

The IR REMOTE board is a small, 2 layer, 1.690" x 1.060", 0.031" thick PCB with SMT components on both sides (see the IVEX WinBoard PCB file → irremote.brd, as well as the GERBER files). The PCB has 0.012" minimum trace width, 0.006" minimum spacing, and 0.019" minimum pitch on SMT pads. The layout of the board and many of the components chosen were dictated by the Polycase FB-20 enclosure . The choice of the PW (0.65 mm pitch spacing) package for the 20-pin MSP430F11X1A and the use of the 10-pin 0.5mm pitch flat flex cable (FFC) connector as the JTAG connector are a direct result of this. The layout was further complicated by fixed locations for certain components – coin cell battery holder, elastomer button keypad targets, and the 3mm T-1 IR LED (a larger T-1¾ IR LED would not fit in the package) (see IR REMOTE CONTROL UNIT ASSEMBLY, IR1001.DWG and ENCLOSURE MODIFICATION OF FB-20 FOR T-1 LED, IR0101.DWG). An additional complication in the layout was in trying to keep it to only 2 layers (to keep the cost down). The routing and layout was done manually. The components are oriented (for example the JTAG connector) for routing and not for ease of use – no vias could be put in the area where the keypad targets were (see IR REMOTE BOARD ASSEMBLY Drawing, IR0001.dwg). The minimum trace width, minimum spacing, PCB thickness, as well as the PCB shape all conspired to make this board unsuitable for all of the low cost PCB fabrication specials. I did not have the keypad targets gold plated, but this may be something to consider for an actual product application.



IR REMOTE Board (Populated and Unpopulated, Both Sides)

Component details and pricing are in the bill of materials (BOM) (see IR Remote BOM, IRREMOTE_BOM.xls). The component cost for the IR REMOTE unit is $21.54 each in quantities of 20 of which $11.00 is for the PCB itself. The PCB itself goes down to the sub $2 range in quantities of several hundred (it is probably cheaper to go with a lot charge than for individual numbers of boards). In quantities of 20, the Polycase FB-20 enclosure (any version with the elastomer keypad) is only $2.05.

If you have a steady hand, a well lighted anti-static work station, a large magnifying glass, lots of patience, and a sharp soldering iron tip, you can hand assemble and hand solder the IR REMOTE board. I managed to burn the tips of my fingers a number of times and still successfully build several boards. I would strongly suggest that you find a local PCB assembly house that specializes in small runs to assemble the boards for you if you need more than a handful. The PCB assembly house is likely to do a better job plus save your fingertips and your sanity.

The software is designed to allow developers to easily modify it to meet their own needs/requirements. The software is designed to transmit a separate IR PCM code on each button press and each button release. With five buttons up to 10 separate codes can be transmitted. The PCM code used is up to the developer. The only requirement is that the code be compatible with the Vishay TSOP1833SS3V receiver and the receiver software. In this application, each button press and each button release will transmit a unique 8-bit code. This could easily be changed to unique 16-bit, 24-bit, or 32-bit codes, allowing maximum customization. Clearly this approach is not suitable for real security applications such as door locks, etc. It does, however, allow a developer to provide unique solutions of the same IR REMOTE to different clients, for different product models, even all the way to different end users (if you wanted to go to that much trouble).

The PCM protocol used is a 32.768 kHz IR burst for 2ms as a start sentinel with a 1.5ms burst as a "1" and a 1ms burst as a "0". There is a 1ms break (with no IR transmission) following each IR burst.

```
_||||__||__|||__||__||__|||__||__|||__|||__
 "S"  "0"  "1"  "0"  "0" "1"  "0"  "1"  "1"
     MSB -------------to--------------- LSB
```

"S" ➔ start sentinel, "__" ➔ space, "||||, |||, ||" ➔ IR burst transmissions

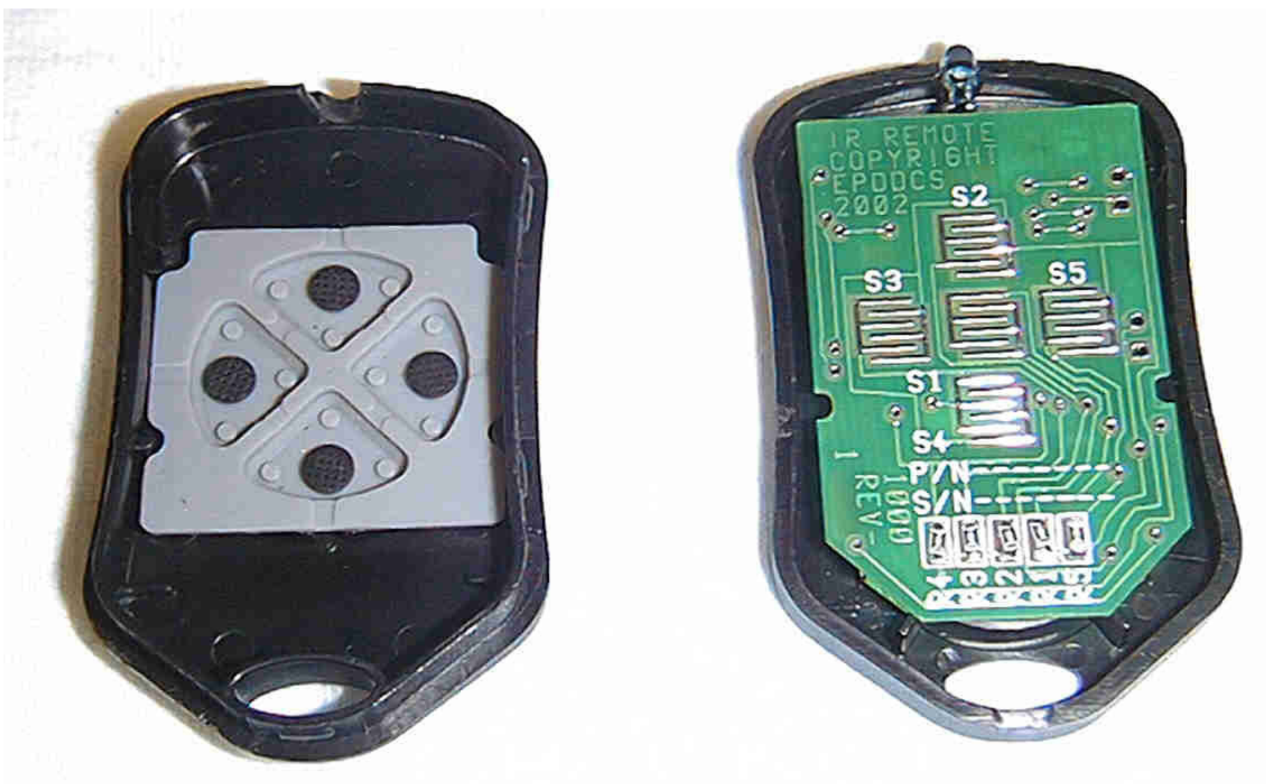
The Vishay TSOP1833SS3V PCM receiver chip will output the following digital serial signal in response to the above IR PCM transmission:

-____--__--___--__--__--___--__--___--___-- "-" = 3V, "_" = 0V
“S”   “0” “1” “0” “0”   “1” “0” “1” “1”

The receiver uC interfacing with the Vishay TSP1833SS3V PCM receiver chip must time the active logic low intervals as well as the logic high space intervals to determine if a valid IR REMOTE command code has been received. If the received signal does not match the PCM protocol or the list of codes in the receiver, the signal should be ignored. The developer can also modify this simple PCM protocol to whatever he or she might like by just changing the IR burst periods or the space period and/or the definitions of each.

The IR REMOTE along with the Polycase TB-20 enclosures, expects the operator to hold the IR REMOTE in one hand and press a button with the thumb of that same hand -- the same way you operate a keyless entry key fob to open a car. The software assumes that only one valid key press will be made at a time. If two (or more) button presses are made at the same instant (within a few uS), then the software will assume that only the lowest button press is legitimate and ignore any higher button. Any button pressed after the interrupt will be ignored until the legitimate button pressed is released.

The IR REMOTE buttons are numbered in the following manner (looking down on the IR REMOTE as you would hold it to operate it): S1 is in the center and the rest of the buttons are numbered counter clockwise around S1 starting with S2 which is closest to the IR LED.



IR REMOTE Unit Open (Note Elastomer Button Contact Pads and Switch Numbering)

The software has two modes: TEST and NORMAL. TEST Mode is entered if a button is pressed when the unit comes out of power on reset (POR). TEST Mode is an endless loop where the unit continuously transmits the button codes in a sequential manner with a ½ second delay between codes. TEST Mode is used primarily for testing the IR receiver. In TEST mode the unit does not go to sleep and the buttons are ignored. The only way out of TEST Mode is to reset the unit.

NORMAL MODE is the system standard operating mode. The system will end up in NORMAL MODE following a POR if no buttons were pressed as the unit came out of POR. The system will stay in this loop forever. The button keypad inputs to the MSP430 are interrupt driven and not scanned.

The system will initialize itself to look for button press (falling edge) interrupts on P2.1-P2.5 inputs, enable interrupts, go into LPM3 sleep mode, and wait for a button press. The first button press interrupt detected (or the one associated with the lowest Port2 I/0 pin if more than one occur together) will wake up the system, disable additional interrupts, and start a debounce delay. Following the debounce delay, the input level of the Port2 pin responsible for the interrupt is read to validate the button press. If not a valid button press, interrupts are enabled again and the system returns to LPM3 sleep and awaits the next button press. If it was a valid button press, then the button code associated with this particular button press is transmitted.

The system will then focus on detecting the release of this particular button to the exclusion of all other button presses and releases. The level of the particular Port2 pin is read again this time looking for a high, indicating a button release. If the input level looks like a button release has already occurred, then the debounce delay is started, followed by rereading the input level. If the button release was not valid or the initial check of the input level did not indicate a button release had occurred yet, then the Port2 interrupt enable and interrupt edge select registers are set up to look only for a button release interrupt on this specific input pin. The system then enables interrupts, clears the port2flag, and goes back to LPM3 sleep. If the button release was valid, the system transmits the specific button release code. The system then reinitializes the Port2 interrupt registers to look for button presses on P2.1-P2.5, clears port2flag, enables interrupts, and goes to LPM3 sleep. The system is now ready for the next button press.

If the system is armed and looking only for a specific button release (following a specific valid button press), only a rising edge interrupt on that specific input can wake up the unit. When that interrupt occurs, the system disables all interrupts, wakes up, and starts the debounce delay. The input level is read to confirm the valid button release. If not confirmed, port2flag is cleared, interrupts are enabled, the system goes back to LPM3 sleep, and waits for the button release. If the button release was confirmed, the system transmits the specific button release code. The system then initializes the Port2 interrupt registers for falling edge (button presses), enables interrupt requests on P2.1-P2.5, enables interrupts, and goes to LPM3 sleep to wait for the next button press.

For more detail about the software, the source code listing in “C” is appended at the end of this document.

The IR transmission time for a key press (or release) code is a function of the length of the code. The minimum transmission time will be for a transmission of all "0"'s and the maximum transmission time will be for a transmission of all "1"'s. The formulas for transmission times are:

min transmission time = 3ms + ((# of bits) x 2ms)
max transmission time = 3ms + ((# of bits) x 2.5ms)
3ms = start sentinel + space = 2ms + 1ms = 3ms
2ms = "0" + space = 1ms + 1ms = 2ms
2.5ms = "1" + space = 1.5ms + 1ms = 2.5ms

# bits in code	min	max
8	19ms	23ms
16	35ms	43ms
24	51ms	63ms
32	67ms	83ms

If the debounce delay is added to the maximum transmission times, then worst case times can be determined from the first press (or release) of a button until the completion of the IR transmission of that button's code. The debounce delay is 30ms.

# bits in code	max time from key press

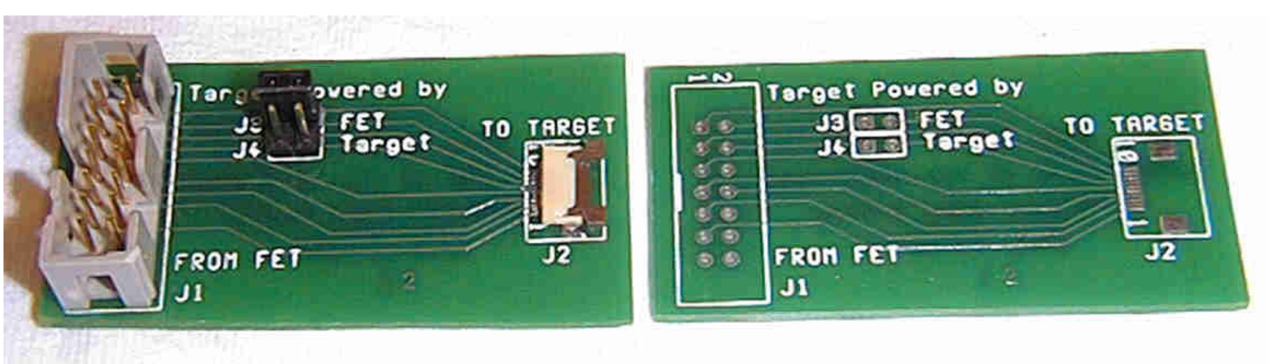
	to end of IR transmission
8	53ms
16	73ms
24	93ms
32	113ms

The Vishay TSOP1833SS3V PCM receiver chip has only a few limitations that might effect this application. One limitation is that an IR burst transmission at ~33kHz must last longer than 6 cycles (>182us) and that there must be a period of at least 25ms of no IR transmission in every 150ms. The duration of the start sentinel (2ms), "0" (1ms), and "1" (1.5ms) IR bursts easily meet the first criteria along with the 1ms space between bursts. The second criteria is easily met with the debounce delay as long as key presses/releases are handled as indicated in the software -- one valid key press/release at a time. If the key presses/releases were buffered instead (allowing multiple button presses at the same time), the 25ms of no transmission would need to be enforced every so often depending on the current depth of the buffer, the number of bits of code, and how long since the last 25ms or longer of no transmission.

The current software can recognize a valid button press (or release) and respond almost 19 times a second (with an 8-bit code). This is pressing and releasing a button with your thumb 9 times a second. Even with a 32-bit code, the software can recognize a valid button press (or release) and respond 8 times a second (4 separate button presses and releases a second). Although this technique may not be suitable for typing, it is more than adequate for this application.

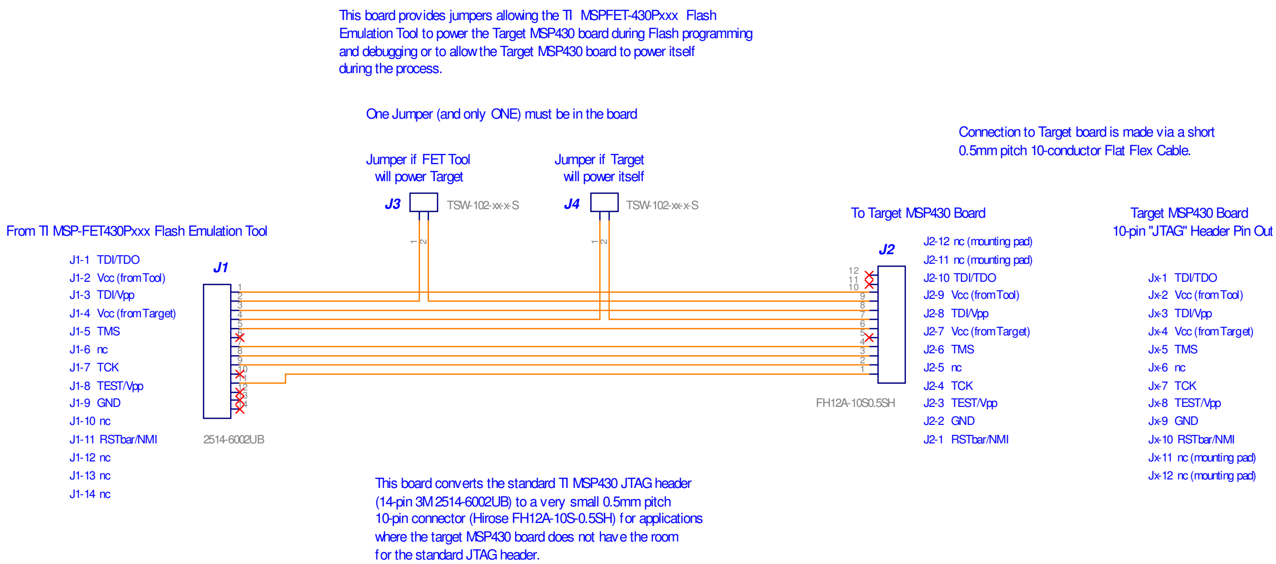
Battery power is not monitored by the MSP430. As the 3V Li coin cell battery (CR2032) discharges over time from 3V to 2V, the current available to the IR LED will decrease. This will result in the IR LED getting dimmer and dimmer during its burst transmissions. The net result is that for the receiver to recognize the transmitter, the transmitter will need to get closer and closer to the receiver as the battery becomes discharged. This is the same thing you experience with your TV remote control as its batteries die out, and the end user will recognize this problem for what it is. The IR REMOTE is designed to last many, many years on the Li coin cell battery. A conservative estimate of 50 complete button press and release cycles (100 separate IR transmissions) per day with a 32-bit code result in an average current consumption for the IR REMOTE of < 1.5 uA (< 1.2 uA with an 8-bit code)! This translates into a CR2032 battery life for the IR REMOTE of >10 years! In reality, the IR REMOTE represents a negligible load to the CR2032 battery and actual battery life is more a function of the ambient temperature and self-discharge than this load.

There is not enough physical space in this application for the standard TI JTAG connector, so the IR REMOTE PCB has a 10-pin 0.5mm pitch flat flex cable (FFC) connector. An additional PCB was designed to convert from the standard TI 14-pin JTAG connector to a 10-pin 0.5mm pitch FFC connector, which along with a short 10-conductor 0.5mm pitch FFC cable allows the TI FET to connect to the IR REMOTE board for Flash programming/debugging (see the IVEX WinDraft schematic file ➔ con14to10_a.sch). None of the MSP430 JTAG pins on the IR REMOTE design are shared. The JTAG 14-pin to 10-pin Board has two 2-pin headers, J3 & J4, for a jumper to let the FET know whether the FET is powering the target or whether the target is powering itself. Normally, these jumpers are on the target, but there was no room.



JTAG 14-pin to 10-pin Connector Board (Populated and Unpopulated)

The JTAG 14-pin to 10-pin Connector Board is a 2 layer, 7.200” x 2.200”, 0.062” thick PCB with thru hole and SMT components on one side (see the IVEX WinBoard PCB file → con14-10.brd, as well as the GERBER files). The PCB has 0.012” minimum trace width, 0.006” minimum spacing, and 0.019” minimum pitch on SMT pads. Unlike the IR REMOTE, this PCB would definitely qualify for the low cost PCB fabrication specials. To maximize the return, the board file has 6 identical copies of the circuit in the schematic. The reference designators change for each circuit (since the low cost PCB fabrication specials exclude panelization), essentially ending up with a pseudo legal “poor man’s” panelization scheme. You have to cut each PCB into the six pseudo identical boards yourself – this is what accounts for the slightly different sizes of the two boards in the picture above. The BOM (see JTAG 14-pin to 10-pin Connector BOM, con14to10.xls) has the reference designators correctly identified along with component details and pricing. The JTAG 14-pin to 10-pin Board has a parts cost of $14.66 each in quantities of 12. $8.35 of this is the cost of the PCB itself. You only need one JTAG 14-pin to 10-pin Connector Board for development and programming of the IR REMOTE or for any other MSP430 application where you don’t have room for the 3M JTAG connector favored by TI.

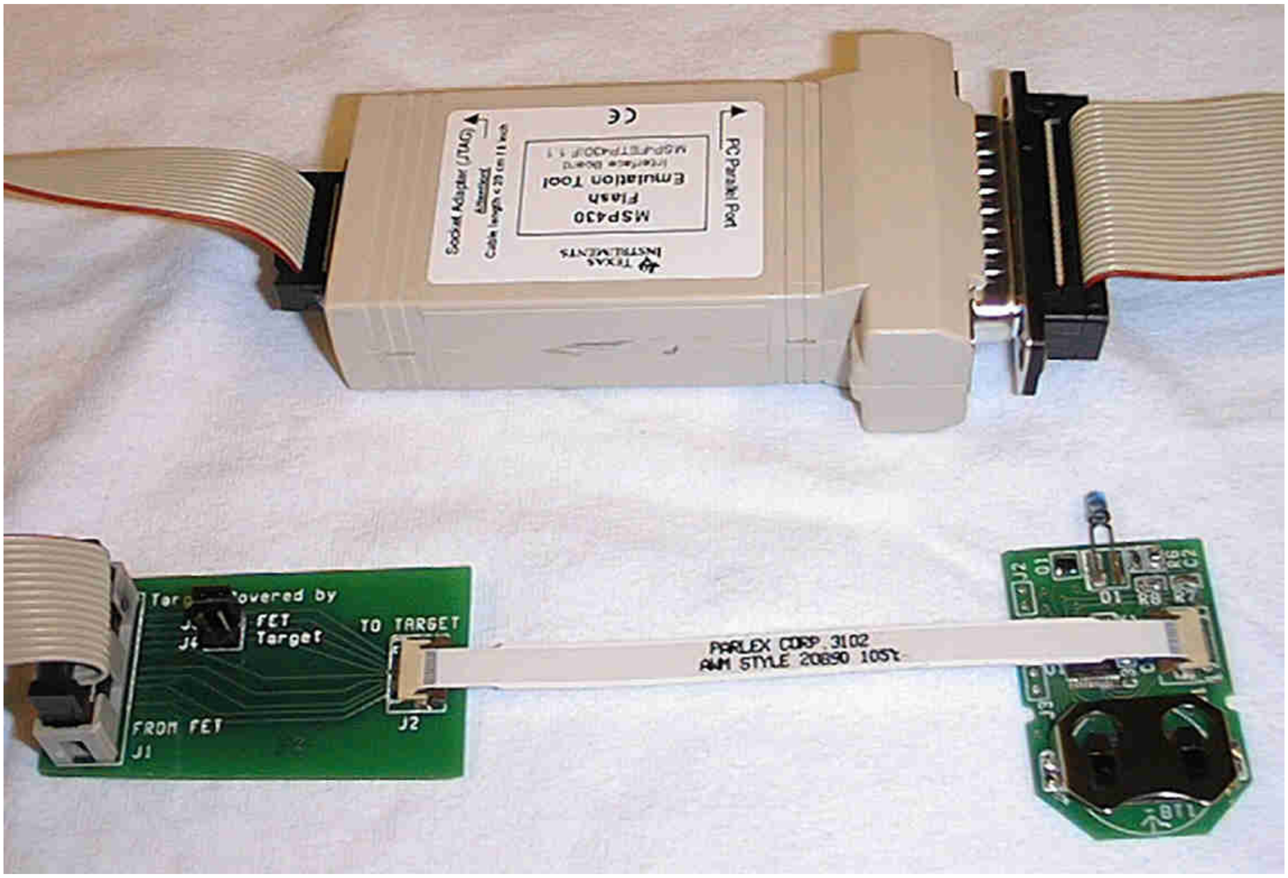


JTAG 14-pin to 10-pin Connector Circuit

The software was developed with the IAR Embedded Workbench for the MSP430 version: 1.25A, along with the TI MSP430 FET and an Olimex MSP430-H1121 (MSP430F1121) header board. In addition an HP 54645D Mixed Signal Oscilloscope was also used in debugging. If the physical size of the IR REMOTE PCB itself and the pitch of the 20-pin MSP430 PW package had not been so small, it could have been used as the development target.

The only internal peripheral used in this application is the 16-bit timer, Timer_A. The 32.768 kHz ACLK is the input to Timer_A to time the debounce delay (30ms) as well as the various intervals in the IR transmission -- start sentinel (2ms), "1" (1.5ms), "0" (1ms), and space (1ms). The on chip DCO with the on chip resistor is used as MCLK.

The watchdog is disabled and not used in this application.



JTAG Tool Chain to Program & Debug Code on the IR REMOTE

Although development was done on the MSP430F1121, the IR REMOTE board is layed out for any MSP430x11xIPW or MSP430x11x1(A)IPW device. The application turns out to be 1466 bytes, and will run on a MSP430F1111AIPW with 128 bytes of RAM and 2k of FLASH. The software is not optimized to try and fit it into a 1k device (MSP430F1101AIPW or MSP430F110IPW), primarily to keep the software easy for a developer to follow, understand, and modify. (NOTE: If your application does not need codes transmitted on both the pressing and releasing of each button (i.e., code transmission ONLY occurs on button presses for all buttons or ONLY on button releases for all buttons) AND you must fit the firmware into 1k without resorting to tuning (possibly in assembly), the code can be modified to easily fit.)

The intent of this application is to provide an easy to modify reference design where a developer can produce a customized solution very easily and very rapidly, primarily by just modifying the button codes. The IR REMOTE is targeted at applications such as relatively small production runs and/or professional looking prototypes, both of which can not justify the nonrecurring engineering costs (NRE's) for a custom injection-molded remote control, but still need a professional looking solution. These applications typically cannot justify the additional engineering development effort to produce relatively meager (~$1) per unit cost savings such as might be found by going to an MSP430 version with slightly less memory.

Unless you are looking at serious numbers of IR REMOTES, there is not that much additional cost savings to be had. The largest savings are from going with a ROM part and removing the small 10-pin JTAG header. You may also be able to parallel a number of the unused I/O pins to drive the IR LED directly and get rid of the MOSFET. This was not attempted due to the need for as much drive as quickly as possible so the IR LED would be on and as bright as possible during the 15us the LED is on at a time during the 32.768kHz transmission bursts. Since the MSP430's I/O output levels are a function of load, it was not clear how many paralleled output pins were needed. The on-resistance of the MOSFET is both spec'ed and very low -- neither of which is true of the MSP430 I/O pins.

REFERENCES

1. Texas Instruments MSP430C11x1, MSP430F11x1A Mixed Signal Microcontroller Data Sheet SLAS241E (Revised July 2002)

2. Texas Instruments MSP430x1xx Family User's Guide SLAU049A (2001)

3. TI MSP430 Application Report SLAA134 (Sept 2001) -- "Decode TV IR Remote Control Signals Using Timer_A3"

4. TI MSP430 Family Application Report SLAAE10C (March 1998) – Section 4.10.3 Remote Control Applications

5. Texas Instruments MSP-FET430 Flash Emulation Tool (FET) User's Guide Version 3.03 (2/12/02)

6. TI MSP430 website (www.ti.com) ==> MSP430 ==> example code

7. IAR Application Note 430-02 (Sept 99) -- MSP430 Power-Down Modes

8. IAR Application Note 430-04 -- Disabling and enabling interrupts in interrupt-disabled environments for MSP430

9. VISHAY Optoelectronics/Infrared Receiver Modules (TSOP18..SS3V Photo Modules for PCM Remote Control Systems) & IR Emitting Diodes (TSAL4400 GaAs/GaAlAs IR Emitting Diode in 3mm (T-1) Package) (www.vishay.com) Also see application notes/background info on these parts and on infrared pulse code transmission techniques.

10. Polycase Plastic Electronic Enclosures, FB-20 series 1 to 5 button wireless remote enclosure (www.polycase.com)

11. RENATA Lithium Batteries Designer's Guide (www.renata.com), CR2032 discharge curves & pulsed duty curves

12. Agilent Technologies Application Note 1395 -- Debugging Serial Bus Systems with a Mixed-Signal Oscilloscope

IR REMOTE PROJECT RELATED DOCUMENTS

1. MSP430 Based IR Remote Control, rev -, 8/14/02, schematic file ==> IR_REMOTE.SCH IVEX WinDraft v3.10 (see www.ivex.com for tool info)

2. IRREMOTE.BRD, rev -, 8/7/02, PCB file, IVEX WinBoard v2.25 (see www.ivex.com for tool info) (matches schematic file above) (All PCB fabrication files: Gerber, Drill, Dimensions, Pick & Place, etc., are generated from this PCB file.)

3. IR Remote Board Assembly Drawing, rev -, C size, IR0001.DWG (AutoCAD format)

4. IR Remote Control Unit Assembly Drawing, rev -, C size, IR1001.DWG (AutoCAD format)

5. Enclosure Modification of FB-20 for T-1 LED Part Drawing, rev -, C size, IR0101.DWG (AutoCAD format)

6. IR Remote Control BOM, rev -, IRREMOTE_BOM.XLS (MS EXCEL format)

7. JTAG 14-pin to 10-pin Connector, rev -, 8/16/02, schematic file ==> CON14TO10.SCH IVEX WinDraft v3.10 (see wwwlivex.com for tool info)

8. CON14-10.BRD, rev -, 8/17/02, PCB file, IVEX WinBoard v2.25 (see www.ivex.com for tool info) (This board file makes 6 copies of the board, but with different reference designators -- low budget panelization. The schematic and netlist above match only the first circuit of the six.) (All PCB fabrication files: Gerber, Drill, Dimensions, Pick & Place, etc., are generated from this PCB file.)

9. JTAG 14-pin to 10-pin Connector Board Assembly Drawing, rev -, A size, IR0002.DWG (AutoCAD format)

10. JTAG 14-pin to 10-pin Connector BOM, rev -, CON14TO10.XLS (MS EXCEL format)

```
//*****************************************************************************
//
//          IR REMOTE -- Infrared Remote Control Reference Design
//
//          SOURCE CODE FILE ==>    ir_remote_c.c
//
//*****************************************************************************
//          IR REMOTE -- Infrared Remote Control Reference Design
//
//    IR REMOTE is an IR Pulse Code Modulation (PCM) Remote Control
//    Reference Design built around the TI ultra low-power MSP430 uC,
//    Vishay TSOP1833SS3V 33.0 kHz Photo Module for PCM Remote Control Systems,
//    Vishay TSAL4400 Infrared Emitter, 3V Li coin cell battery (CR2032),
//    and the 1 to 5-button key fob style Polycase TB-20 Enclosures. This
//    reference design will allow a professional, low-cost, small, battery-
//    powered, and easily customized IR remote control capability to be added
//    to a customer's (or hobbyist's) project.
//
//    There are many applications where a small and relatively simple remote
//    control might be used:  light dimmers, overhead fan controls,
//    automated blinds/awnings, heating/air conditioning controls, data
//    acquisition control, instrumentation control, and a wide variety of
//    difficult to access industrial sensors/controls, etc.
//
//    IR REMOTE is targeted at any application where ON/OFF, UP, DOWN,
//    MODE, BRIGHTER, DIMMER, FASTER, SLOWER, etc., type of IR remote control
//    commands would make sense, plus where a remote control package needs
//    to have a professional and non-hobbyist appearance.
//
//    The Polycase TB-20 enclosures and their elastomer keypads are
//    inexpensive even in small numbers (1-10 units) and can be customized
//    (screen printed, painted, and/or labeled) for far less than having
//    custom membrane keypads/overlays made, or having custom elastomer
//    keypads made, or having custom injection molded plastic cases made.
//
//    The IR REMOTE control is a TI MSP430F1111IPW or MSP430F1121IPW
//    based design powered by a 3V Li coin cell (CR2032).  There are five
//    elastomer button keypad targets on the PCB that match all of the possible
//    1 to 5 button keypad versions of the Polycase TB-20 enclosures.  All
//    five of the keypad button inputs are pulled up to 3V via 1M resistors.
//    Pressing an elastomer button will cause the keypad button input to
//    be pulled down to ground.
//
//    The keypad button inputs are:
//                              S1 on P2.1
//                              S2 on P2.2
//                              S3 on P2.3
//                              S4 on P2.4
//                              S5 on P2.5.
//
//    The only output is P2.0.  P2.0 drives an N-channel MOSFET that in turn
//    drives the 3mm Infrared LED (Vishay TSAL4400).  The IR LED is driven at
//    32.768kHz, which is conveniently very close to the center of the IR
//    receiver's 33.0 kHz (+/- 5%) carrier frequency.  If the MSP430 has an
//    external 32.768 kHz low frequency crystal, then the IR LED modulation
//    can easily be generated from the 32.768 kHz ACLK.  The IR LED will draw
//    just under 15mA when ON (this is a design limitation imposed by the
//    pulsed duty limits of the CR2032 coin cell battery).  The IR REMOTE
//    is designed to work with a Vishay TSOP1833SS3V 33.0 kHz PCM 3-pin
//    (power, ground, and output) receiver chip incorporated into the
//    design of the product being controlled.  The output of the TSOP1833SS3V
//    is a digital serial signal matching the infrared PCM input, but with
//    the carrier removed (active low logic signal = carrier, high = no
//    carrier).  The TSOP1833SS3V is a sophisticated part with considerable IR
//    noise filtering and immunity, resulting in robust and reliable PCM
//    reception.  Due to the pulse duty limitation of the 3V Li coin cell
//    battery, the IR REMOTE will only work line of sight and will need to be
//    pointed in the direction of the receiver.  (Your TV remote
//    control pulses its IR LED at a much higher current due to its alkaline
//    battery(s) and is able to saturate a room with its IR signal so it does
//    not need direct line of sight to work with the TV.)
//
//    The software is designed to allow reference users to easily modify
//    it to meet their own needs/requirements.  The software is designed to
//    transmit a separate IR PCM code on each button press and each button
//    release.  With five buttons up to 10 separate codes can be transmitted.
```

```
//    The PCM code used is up to the developer.  The only requirement is that
//    the code be compatible with the Vishay TSOP1833SS3V receiver and the
//    receiver software.  In this application, each button press and each button
//    release will transmit a unique 8-bit code.  This could easily be changed
//    to unique 16-bit, 24-bit, or 32-bit codes, allowing maximum customization.
//    Clearly this approach is not suitable for real security applications
//    such as door locks, etc.  It does, however, allow a developer to provide
//    unique solutions of the same IR REMOTE to different clients, for
//    different product models, even all the way to different end users (if
//    you wanted to go to that much trouble).
//
//    The PCM protocol used is a 32.768 kHz IR burst for 2ms as a start
//    sentinel with a 1.5ms burst as a "1" and a 1ms burst as a "0".  There is
//    a 1ms break (with no IR transmission) following each IR burst.
//
//        _||||__||__|||__||__||__|||__||__|||__|||__
//          "S" "0"  "1"  "0" "0" "1" "0"  "1"  "1"
//              MSB ----------to--------------- LSB
//
//          "S" = start sentinel, "__" = space,
//          "||||, |||, ||" = IR burst transmission
//
//    The Vishay TSOP1833SS3V PCM receiver chip will output the following
//    digital serial signal in response to the above IR PCM transmission:
//
//        -____--__--___--__--__--___--__--___--___--  "-" = 3V, "_" = 0V
//
//    The uC interfacing with the Vishay TSP1833SS3V PCM receiver chip in the
//    receiver must time the active logic low intervals as well as the logic
//    high space intervals to determine if a valid IR REMOTE command code has
//    been received.  If the received signal does not match the PCM protocol or
//    the list of codes in the receiver, the signal should be ignored.  The
//    developer can also modify this simple PCM protocol to whatever he or she
//    might like by just changing the IR burst periods or the space period
//    and/or the definitions of each.
//
//    The button keypad inputs to the MSP430 are not scanned, but interrupt
//    driven.  The inputs are configured to generate a Port 2 I/O interrupt
//    when a button is pressed.  The ISR will wake up the MSP430 from LPM3
//    sleep.  The MSP430 will then turn off Port 2 I/O interrupts, determine
//    the pressed button(s), and start a debounce delay timer.  After the
//    debounce delay, the MSP430 will scan the input level of the lowest
//    pressed button (lowest, S1, to highest, S5).
//
//    The IR REMOTE along with the Polycase TB-20 enclosures, expects the
//    operator to hold the IR REMOTE in one hand and press a button with the
//    thumb of that same hand -- the same way you operate a keyless entry
//    key fob to open a car.  The software assumes that only one valid key
//    press will be made at a time.  If two (or more) button presses are made
//    at the same instant (within a few uS), then the software will assume
//    that only the lowest button press is legitimate and ignore any higher
//    button.  Any button pressed after the interrupt will be ignored until
//    the legitimate button press is released.
//
//    The IR REMOTE buttons are numbered in the following manner (looking down
//    on the IR REMOTE as you would hold it to operate it):
//                               ^
//                     _______*IR LED*_______
//                    |          ^           |
//                    |                      |
//                    |        ( S2 )        |
//                    |                      |
//                    | ( S3 ) ( S1 ) ( S5 ) |
//                    |                      |
//                    |        ( S4 )        |
//                    |                      |
//                    |______________________|
//
//    If the input level matches a valid key press, then the
//    MSP430 will transmit the corresponding key press code.  After the
//    transmission, the MSP430 will again scan the input level of the key to
//    determine if the button is still pressed or if it has been released during
//    the button press transmission.  If the button is still pressed, the Port
//    2 I/O interrupts are cleared along with the button keypad interrupt
//    enables.  Only the pressed button keypad interrupt enable is enabled and
//    it is configured for a button release (rising edge) interrupt.  The
```

```
//    MSP430 goes back to LPM3 sleep and waits for the button to be released.
//    If the input level of the valid pressed button looks like the button
//    has been released during the IR transmission (and the edge was missed
//    because interrupts were disabled), the MSP430 starts the debounce delay
//    timer.  After the debounce delay, the MSP430 will rescan the input level
//    of the keypad input of interest.  If a valid button release has not
//    occurred, then the software goes through the same procedure above to look
//    for the button release.  If a valid button release has occurred, then
//    the MSP430 will transmit the corresponding key release code.  After the
//    transmission, the MSP430 will configure all keypad inputs for falling
//    edge interrupts, clear all interrupts, enable interrupts, and go to LPM3
//    sleep.
//
//    The IR transmission times for a key press (or release) code is a function
//    of the length of the code.  The minimum transmission time will be for
//    a transmission of all "0"'s and the maximum transmission time will be for
//    a transmission of all "1"'s. The formulas for transmission times are:
//          min transmission time = 3ms + ((# of bits) x 2ms)
//          max transmission time = 3ms + ((# of bits) x 2.5ms)
//            3ms = start sentinel + space = 2ms + 1ms = 3ms
//            2ms = "0" + space = 1ms + 1ms = 2ms
//            2.5ms = "1" + space = 1.5ms + 1ms = 2.5ms
//
//                # bits in code    min     max
//                    8             19ms    23ms
//                    16            35ms    43ms
//                    24            51ms    63ms
//                    32            67ms    83ms
//
//    If the debounce delay is added to the maximum transmission times, then
//    worst-case times can be determined from the first press (or release) of
//    a button until the completion of the IR transmission of that button's
//    code.  Assume the debounce delay is 30ms.
//
//              # bits in code    max time from key press
//                                to end of IR transmission
//                    8                   53ms
//                    16                  73ms
//                    24                  93ms
//                    32                  113ms
//
//    The Vishay TSOP1833SS3V PCM receiver chip has only a few limitations
//    that might effect this application.  One limitation is that an IR burst
//    transmission at ~33kHz must last longer than 6 cycles (>182us) and that
//    there must be a period of at least 25ms of no IR transmission in every
//    150ms.  The duration of the start sentinel (2ms), "0" (1ms), and "1"
//    (1.5ms) IR bursts easily meet the first criteria along with the 1ms
//    space between bursts.  The second criteria is easily met with the
//    debounce delay as long as key presses/releases are handled as indicated
//    in the software -- one valid key press/release at a time.  If the key
//    presses/releases were buffered instead (allowing multiple button presses
//    at the same time), the 25ms of no transmission would need to be enforced
//    every so often depending on the current depth of the buffer, the number
//    of bits of code, and how long since the last 25ms or longer of no
//    transmission.
//
//    The current software can recognize a valid button press (or release) and
//    respond almost 19 times a second (with an 8-bit code).  This is pressing
//    and releasing a button with your thumb 9 times a second.  Even with a
//    32-bit code, the software can recognize a valid button press (or release)
//    and respond 8 times a second (4 separate button presses and releases a
//    second).  Although this technique may not be suitable for typing, it
//    is more than adequate for this application.
//
//    Battery power is not monitored by the MSP430.  As the 3V Li coin cell
//    battery (CR2032) discharges over time from 3V to 2V, the current
//    available to the IR LED will decrease.  This will result in the IR LED
//    getting dimmer and dimmer during its burst transmissions.  The net result
//    is that for the receiver to recognize the transmitter, the transmitter
//    will need to get closer and closer to the receiver as the battery
//    becomes discharged.  This is the same thing you experience with your
//    TV remote control as its batteries die out, and the end user will
//    recognize this problem for what it is.  The IR REMOTE is designed to last
//    many, many years on the Li coin cell battery.  A conservative estimate of
//    50 complete button press and release cycles (100 separate IR
//    transmissions) per day with a 32-bit code result in an average current
```

```
//    consumption for the IR REMOTE of < 1.5 uA (< 1.2 uA with an 8-bit code)!
//    This translates into a CR2032 battery life for the IR REMOTE of
//    >10 years!  In reality, the IR REMOTE represents a negligible load to the
//    CR2032 battery and actual battery life is more a function of the ambient
//    temperature and self-discharge than this load.
//
//    There is not enough space in this application for the standard TI JTAG
//    connector, so the IR REMOTE PCB has a 10-pin 0.5mm pitch FFC connector.
//    An additional PCB was designed to convert from the standard TI 14-pin JTAG
//    connector to a 10-pin 0.5mm pitch flat flex cable (FFC) connector, which
//    along with a short 10-conductor 0.5mm pitch FFC cable allows the TI FET
//    to connect to the IR REMOTE board for Flash programming/debugging.  None
//    of the MSP430 JTAG pins on the IR REMOTE design are shared.
//
//    The software was developed with the IAR Embedded Workbench for the MSP430
//    version: 1.25A, along with the TI MSP430 FET and an Olimex MSP430-H1121
//    (MSP430F1121)header board.  In addition an HP 54645D Mixed Signal
//    Oscilloscope was also used in debugging.  If the physical size of the IR
//    REMOTE PCB itself and the pitch of the 20-pin MSP430 PW package had not
//    been so small, it could have been used as the development target.
//
//         MSP430F1111AIPW
//         MSP430F1121AIPW
//        _________________
//     /|\|            XIN|---
//      | |               | 32.768kHz
//      --|RST        XOUT|---
//        |               |
//        |           P2.0|--> IR LED (low = OFF, high = LED ON)
//        |           P2.1|<-- S1 (low = button press)
//        |           P2.2|<-- S2 (low = button press)
//        |           P2.3|<-- S3 (low = button press)
//        |           P2.4|<-- S4 (low = button press)
//        |           P2.5|<-- S5 (low = button press)
//
//    The only internal peripheral used in this application is the 16-bit timer,
//    Timer_A.  The 32.768 kHz ACLK is the input to Timer_A to time the
//    debounce delay (30ms) as well as the various intervals in the IR
//    transmission -- start sentinel (2ms), "1" (1.5ms), "0" (1ms), and space
//    (1ms).  The on chip DCO with the on chip resistor is used as MCLK.
//
//    The watchdog is disabled and not used in this application.
//
//    Although development was done on the MSP430F1121, the IR REMOTE board
//    is layed out for any MSP430x11xIPW or MSP430x11x1(A)IPW device.  The
//    application turns out to be 1466 bytes, and will run on a
//    MSP430F1111AIPW with 128 bytes of RAM and 2k of FLASH.  The software
//    is not optimized to try and fit it into a 1k device (MSP430F1101AIPW or
//    MSP430F110IPW), primarily to keep the software easy for a developer
//    to follow, understand, and modify.
//
//    NOTE: If your application does not need codes transmitted on both the
//    pressing and releasing of each button (i.e., code transmission ONLY
//    occurs on button presses for all buttons or ONLY on button releases
//    for all buttons) AND you must fit the firmware into 1k without resorting
//    to tuning (possibly in assembly), the code can be modified to easily fit.
//
//    The intent of this application is to provide an easy to modify reference
//    design where a developer can produce a customized solution very easily
//    and very rapidly, primarily by just modifying the button codes.  The IR
//    REMOTE is targeted at applications such as relatively small production
//    runs and/or professional looking prototypes, both of which can not
//    justify the nonrecurring engineering costs (NRE's) for a custom
//    injection-molded remote control, but still need a professional
//    looking solution.  These applications typically cannot justify the
//    additional engineering development effort to produce relatively meager
//    (~$1) per unit cost savings such as might be found by going to an MSP430
//    version with slightly less memory.
//
//    Unless you are looking at serious numbers of IR REMOTES, there is not
//    that much additional cost savings to be had.  The largest savings are
//    from going with ROM parts and removing the small 10-pin JTAG
//    header.  You may also be able to parallel a number of the unused I/O
//    pins to drive the IR LED directly and get rid of the MOSFET.  This was
//    not attempted due to the need for as much drive as quickly as possible so
//    the IR LED would be on and as bright as possible during the 15us the LED
```

```
//    is on at a time during the 32.768kHz transmission bursts.  Since the
//    MSP430's I/O output levels are a function of load, it was not clear how
//    many paralleled output pins were needed.  The on-resistance of the
//    MOSFET is both spec'ed and very low -- neither of which is true of the
//    MSP430 I/O pins.
//
//******************************************************************************
//      REFERENCES
//
//    Texas Instruments MSP430C11x1, MSP430F11x1A Mixed Signal Microcontroller
//      Data Sheet SLAS241E (Revised July 2002)
//
//    Texas Instruments MSP430x1xx Family User's Guide SLAU049A (2001)
//
//    TI MSP430 Application Report SLAA134 (Sept 2001) -- "Decode TV IR
//      Remote Control Signals Using Timer_A3"
//
//    TI MSP430 Family Application Report SLAAE10C (March 1998) -- Section
//      4.10.3 Remote Control Applications
//
//    Texas Instruments MSP-FET430 Flash Emulation Tool (FET) User's Guide
//      Version 3.03 (2/12/02)
//
//    TI MSP430 website (www.ti.com) ==> MSP430 ==> example code
//
//    IAR Application Note 430-02 (Sept 99) -- MSP430 Power-Down Modes
//
//    IAR Application Note 430-04 -- Disabling and enabling interrupts in
//      interrupt-disabled environments for MSP430
//
//    VISHAY Optoelectronics/Infrared Receiver Modules (TSOP18..SS3V Photo
//      Modules for PCM Remote Control Systems) & IR Emitting Diodes
//      (TSAL4400 GaAs/GaAlAs IR Emitting Diode in 3mm (T-1) Package)
//      (www.vishay.com) Also see application notes/background info on these
//      parts and on infrared pulse code transmission techniques.
//
//    Polycase Plastic Electronic Enclosures, FB-20 series 1 to 5 button
//      wireless remote enclosure (www.polycase.com)
//
//    RENATA Lithium Batteries Designer's Guide (www.renata.com), CR2032
//      discharge curves & pulsed duty curves
//
//    Agilent Technologies Application Note 1395 -- Debugging Serial Bus
//      Systems with a Mixed-Signal Oscilloscope
//
//******************************************************************************
//      IR REMOTE PROJECT RELATED DOCUMENTS
//
//    MSP430 Based IR Remote Control, rev -, 8/14/02, schematic file ==>
//      IR_REMOTE.SCH  IVEX WinDraft v3.10 (see www.ivex.com for tool info)
//
//    IRREMOTE.BRD, rev -, 8/7/02, PCB file, IVEX WinBoard v2.25 (see
//      www.ivex.com for tool info) (matches schematic file above) (All
//      PCB fabrication files: Gerber, Drill, Dimensions, Pick & Place, etc.,
//      are generated from this PCB file.)
//
//    IR Remote Board Assembly Drawing, rev -, C size, IR0001.DWG (AutoCAD
//      format)
//
//    IR Remote Control Unit Assembly Drawing, rev -, C size, IR1001.DWG
//      (AutoCAD format)
//
//    Enclosure Modification of FB-20 for T-1 LED Part Drawing, rev -, C size,
//      IR0101.DWG (AutoCAD format)
//
//    IR Remote Control BOM, rev -, IRREMOTE_BOM.XLS (MS EXCEL format)
//
//    JTAG 14-pin to 10-pin Connector, rev -, 8/16/02, schematic file ==>
//      CON14TO10.SCH  IVEX WinDraft v3.10 (see wwwlivex.com for tool info)
//
//    CON14-10.BRD, rev -, 8/17/02, PCB file, IVEX WinBoard v2.25 (see
//      www.ivex.com for tool info) (This board file makes 6 copies of the
//      board, but with different reference designators -- low budget
//      panelization.  The schematic and netlist above match only the
//      first circuit of the six.)  (All PCB fabrication files: Gerber, Drill,
//      Dimensions, Pick & Place, etc., are generated from this PCB file.)
```

```
//
//    JTAG 14-pin to 10-pin Connector Board Assembly Drawing, rev -, A size,
//      IR0002.DWG (AutoCAD format)
//
//    JTAG 14-pin to 10-pin Connector BOM, rev -, CON14TO10.XLS (MS EXCEL
//      format)
//
//******************************************************************************
//
//      Copyright 2002 by
//
//      Murdock Taylor
//      Design Consultant
//      EPDDCS
//      P.O. Box 4739
//      Cary, NC  27519
//      USA
//      (919)460-0081 Voice/FAX
//      EMAIL: murdock.taylor@EPDDCS.com
//
//
//      REVISION HISTORY
//
//      Version:        1.00
//      Tools:          IAR Embedded Workbench for the MSP430 version: 1.25A
//      Released:       10/25/02
//      Comments:       Initial Release -- 1466 bytes
//                      Tested on MSP430F1111AIPW (128 bytes RAM & 2k FLASH) &
//                      MSP430F1121AIPW (256 bytes RAM & 4k FLASH)
//
//******************************************************************************
//
// Software comments assume user is familiar with the MSP430 data sheet, the
// MSP430 header file definitions, and the IAR Embedded Workbench for the MSP430
//
// NOTE:  Some debugging code may have been left in but commented out.
// Debugging code comments are preceded with an "*" ==>  // *Debugging code
//
// unsigned char are 8-bit (byte) with values from 0 to 255
// unsigned int are 16-bit (word) with values from 0 to 65535
// unsigned long are 32-bit (2 words) with values from 0 to 4294967295
// All pointers are 16-bit (word) and can point to memory in
// the range of 0x0000 - 0xFFFF
// NOTE:  The MSP430 operates most efficiently on 16-bit (word) data types
//
// NOTE:  The include header file for the MSP430 version used and the
// project settings in the project file in the IAR Embedded Workbench for
// the MSP430 (i.e. myproject.prj) must match the MSP430 version.


// Include header file for the MSP430 version used

#include  <msp430x11x1.h>


// Include # Defines

#define BBIT0     (0x01)         // Bit 0 (LSB) in an 8-bit byte
#define BBIT1     (0x02)         // Bit 1 in an 8-bit byte
#define BBIT2     (0x04)         // Bit 2 in an 8-bit byte
#define BBIT3     (0x08)         // Bit 3 in an 8-bit byte
#define BBIT4     (0x10)         // Bit 4 in an 8-bit byte
#define BBIT5     (0x20)         // Bit 5 in an 8-bit byte
#define BBIT6     (0x40)         // Bit 6 in an 8-bit byte
#define BBIT7     (0x80)         // Bit 7 (MSB) in an 8-bit byte


// Global Variables

// Codes for each button press and release are entered here.  In
// this version of software, only the least significant 8 bits of
// a particular code is transmitted.  If you do not want any action
// taken on a particular button (i.e. S5 is not present in your
// application or release of S2 is to be ignored), just enter
// a code of 0x0000.  The transmit function will not transmit if
// the button code equals zero.  This array would need to be
```

```
// modified if codes greater than 16-bit are to be used, i.e.
// unsigned long for 32-bit values rather than unsigned int
// for 16-bit values.
const unsigned int buttoncodes[10] =
{ 0x0001,   // S1 pressed  [arrayindex = 0]
  0x0002,   // S2 pressed  [arrayindex = 1]
  0x0003,   // S3 pressed  [arrayindex = 2]
  0x0004,   // S4 pressed  [arrayindex = 3]
  0x0005,   // S5 pressed  [arrayindex = 4]
  0x000A,   // S1 released [arrayindex = 5]
  0x000B,   // S2 released [arrayindex = 6]
  0x000C,   // S3 released [arrayindex = 7]
  0x000D,   // S4 released [arrayindex = 8]
  0x000E    // S5 released [arrayindex = 9]
};

unsigned int arrayindex;          // Array index for buttoncodes[]
unsigned int port2flag;           // Port2 interrupt flag


// Function prototypes

void irtest(void);            // IR transmit/receive test function
void debouncedelay(void);     // 30ms button debounce delay using Timer_A
void delay500ms(void);        // 500ms (1/2 sec) delay using Timer_A
void delay2ms(void);          // 2ms delay using Timer_A
void delay1ms(void);          // 1ms delay using Timer_A
void delay1point5ms(void);    // 1.5ms delay using Timer_A
void transmitstart(void);     // Transmit start sentinel followed by space
void transmitone(void);       // Transmit one followed by space
void transmitzero(void);      // Transmit zero followed by space
void transmit(void);          // Transmit button press/release code


// Main

void main(void)
{
  // Initialization (refer to MSP430 data sheet for POR default settings)
  // Default clock 32.768 kHz = LFXT1 = ACLK, MCLK = SMCLK = default DCO

  // Disable watchdog timer (not used in this application)

  WDTCTL = WDTPW + WDTHOLD; // Disable watchdog timer (watchdog not used)

  // *Initialize I/O Port 1 pins P1.0 - P1.3 as outputs
  // used only for debugging (NOTE: I/O Port Registers are 8-bit)
  //
  //P1DIR |= (BBIT3 + BBIT2 + BBIT1 + BBIT0);   // *P1.0 - P1.3 outputs
  //P1OUT &= ~(BBIT3 + BBIT2 + BBIT1 + BBIT0);  // *P1.0 - P1.3 set to low
  //
  // *Initialize P1.4 as output with SMCLK = DCO
  // P1.4 is used by the FET JTAG, so output can only be seen
  // when FET is released (in C-SPY, under FET Options,
  // Enable "Release JTAG on Go")
  // Used to determine LPM3, P1.4 = DCO = SMCLK ==> awake
  //
  //P1DIR |= BBIT4;             // *P1.4 output
  //P1SEL |= BBIT4;             // *P1.4 = SMCLK = DCO

  // Initialize I/O Port 2 pins
  // (NOTE: I/O Port Registers are 8-bit)
  // P2.1, P2.2, P2.3, P2.4, & P2.5 are inputs by default

  P2DIR |= BBIT0;      // P2.0 output, P2.1 - 2.5 inputs
  P2OUT &= ~BBIT0;     // P2.0 output set to low (IR LED OFF)

  // *****     TEST MODE     *****
  //
  // Check if S1, S2, S3, S4, or S5 is pressed following POR
  // and if so call irtest function and go into test mode.
  // In test mode the unit does not go to sleep and ignores
  // subsequent button presses.  The unit will sequentially
  // cycle thru the array of button codes transmitting a
  // code followed by a 1/2 sec delay and repeating this
  // forever.  A POR with no buttons pressed is necessary
```

```
// to get out of test mode.

if(!(P2IN & BBIT1))      // If S1 pressed
{
  irtest();              // then go into test mode
}
if(!(P2IN & BBIT2))      // If S2 pressed
{
  irtest();              // then go into test mode
}
if(!(P2IN & BBIT3))      // If S3 pressed
{
  irtest();              // then go into test mode
}
if(!(P2IN & BBIT4))      // If S4 pressed
{
  irtest();              // then go into test mode
}
if(!(P2IN & BBIT5))      // If S5 pressed
{
  irtest();              // then go into test mode
}

while(1)   // *****     NORMAL MODE     *****
//
// NORMAL MODE is the system standard operating mode.  The system
// will end up in NORMAL MODE following a POR if no buttons were
// pressed as the unit came out of POR.  The system will stay in this
// loop forever.
//
// The system will initialize itself to look for button press
// (falling edge) interrupts on P2.1-P2.5 inputs, enable interrupts,
// go into LPM3 sleep mode, and wait for a  button press.  The first
// button press interrupt detected (or the one associated with the
// lowest Port2 I/0 pin if more than one occur together) will
// wake up the system, disable additional interrupts, and start
// a debounce delay.  Following the debounce delay, the input
// level of the Port2 pin responsible for the interrupt is read
// to validate the button press.  If not a valid button press,
// interrupts are enabled again and the system returns to LPM3
// sleep and awaits the next button press.  If it was a valid
// button press, then the button code associated with this
// particular button press is transmitted.
//
// The system will then focus on detecting the release of this
// particular button to the exclusion of all other button presses
// and releases.  The level of the particular Port2 pin is read
// again this time looking for a high, indicating a button release.
// If the input level looks like a button release has already
// occurred, then the debounce delay is started, followed by
// rereading the input level.  If the button release was
// not valid or the initial check of the input level did
// not indicate a button release had occurred yet, then the
// Port2 interrupt enable and interrupt edge select registers
// are set up to look only for a button release interrupt on this
// specific input pin.  The system then enables interrupts,
// clears the port2flag, and goes back to LPM3 sleep.  If the
// button release was valid, the system transmits the specific
// button release code.  The system then reinitializes the
// Port2 interrupt registers to look for button presses on
// P2.1-P2.5, clears port2flag, enables interrupts, and goes to
// LPM3 sleep.  The system is now ready for the next button
// press.
//
// If the system is armed and looking only for a
// specific button release (following a specific valid button
// press), only a rising edge interrupt on that specific input
// can wake up the unit.  When that interrupt occurs, the
// system disables all interrupts, wakes up, and starts the
// debounce delay.  The input level is read to confirm the
// valid button release.  If not confirmed, port2flag is cleared,
// interrupts are enabled, the system goes back to LPM3 sleep,
// and waits for the button release.  If the button release
// was confirmed, the system transmits the specific button
// release code.  The system then initializes the Port2 interrupt
// registers for falling edge (button presses), enables interrupt
```

```
// requests on P2.1-P2.5, enables interrupts, and goes to LPM3
// sleep to wait for the next button press.
{
  P2IES = 0x3E;               // Select falling edges on P2.1-P2.5
  P2IE = 0x3E;                // Enable interrupts requests on P2.1-P2.5
  P2IFG = 0x00;               // Clear P2 interrupt flag
  port2flag = 0;              // Initialize Port2 flag
  _EINT();                    // Enable interrupts
  _BIS_SR(LPM3_bits);         // Go to LPM3 sleep

  while(port2flag == 0);          // Wait here until interrupt

  if(port2flag == 1)              // Is S1 pressed?
  {
    debouncedelay();
    if(!(P2IN & BBIT1))           // Validate S1 pressed
    {
      arrayindex = 0;             // Index to S1 pressed code
      transmit();                 // Transmit S1 pressed code
      if(P2IN & BBIT1)            // Is S1 released?
      {
        debouncedelay();
        if(P2IN & BBIT1)          // Validate S1 released
        {
          arrayindex = 5;         // Index to S1 released code
          transmit();             // Transmit S1 released code
        }
        else  // S1 release was not validated after debounce
              // so set up to look only for S1 release
        {
          P2IES &= ~BBIT1;        // Select rising edge on P2.1
          P2IE = BBIT1;           // Enable int request on P2.1 only
          port2flag = 0;          // Clear Port2 flag
          _EINT();                // Enable interrupts
          _BIS_SR(LPM3_bits);     // Enter LPM3 sleep
          while(port2flag == 0);  // Wait until interrupt
          if(port2flag == 1)      // Interrupt from P2.1
          {
            debouncedelay();
            if(P2IN & BBIT1)      // S1 released
            {
              arrayindex = 5;     // Index to S1 released code
              transmit();         // Transmit S1 released code
            }
          }
        }
      }
      else  // S1 not yet released, set up and look for it
      {
        P2IES &= ~BBIT1;          // Select rising edge on P2.1
        P2IE = BBIT1;             // Enable int request on P2.1 only
        port2flag = 0;            // Clear Port2 flag
        _EINT();                  // Enable interrupts
        _BIS_SR(LPM3_bits);       // Enter LPM3 sleep
        while(port2flag == 0);    // Wait until interrupt
        if(port2flag == 1)        // Interrupt from P2.1
        {
          debouncedelay();
          if(P2IN & BBIT1)      // S1 released
          {
            arrayindex = 5;     // Index to S1 released code
            transmit();         // Transmit S1 released code
          }
        }
      }
    }
  }
  if(port2flag == 2)              // Is S2 pressed?
  {
    debouncedelay();
    if(!(P2IN & BBIT2))           // Validate S2 pressed
    {
      arrayindex = 1;             // Index to S2 pressed code
      transmit();                 // Transmit S2 pressed code
      if(P2IN & BBIT2)            // Is S2 released?
      {
```

```
      debouncedelay();
      if(P2IN & BBIT2)           // Validate S2 released
      {
        arrayindex = 6;          // Index to S2 released code
        transmit();              // Transmit S2 released code
      }
      else  // S2 release was not validated after debounce
            // so set up to look only for S2 release
      {
        P2IES &= ~BBIT2;         // Select rising edge on P2.2
        P2IE = BBIT2;            // Enable int request on P2.2 only
        port2flag = 0;           // Clear Port2 flag
        _EINT();                 // Enable interrupts
        _BIS_SR(LPM3_bits);      // Enter LPM3 sleep
        while(port2flag == 0);   // Wait until interrupt
        if(port2flag == 2)       // Interrupt from P2.2
        {
          debouncedelay();
          if(P2IN & BBIT2)       // S2 released
          {
            arrayindex = 6;      // Index to S2 released code
            transmit();          // Transmit S2 released code
          }
        }
      }
    }
    else  // S2 not yet released, set up and look for it
    {
      P2IES &= ~BBIT2;           // Select rising edge on P2.2
      P2IE = BBIT2;              // Enable int request on P2.2 only
      port2flag = 0;             // Clear Port2 flag
      _EINT();                   // Enable interrupts
      _BIS_SR(LPM3_bits);        // Enter LPM3 sleep
      while(port2flag == 0);     // Wait until interrupt
      if(port2flag == 2)         // Interrupt from P2.2
      {
        debouncedelay();
        if(P2IN & BBIT2)       // S2 released
        {
          arrayindex = 6;      // Index to S2 released code
          transmit();          // Transmit S2 released code
        }
      }
    }
  }
}
if(port2flag == 3)               // Is S3 pressed?
{
  debouncedelay();
  if(!(P2IN & BBIT3))            // Validate S3 pressed
  {
    arrayindex = 2;              // Index to S3 pressed code
    transmit();                  // Transmit S3 pressed code
    if(P2IN & BBIT3)             // Is S3 released?
    {
      debouncedelay();
      if(P2IN & BBIT3)           // Validate S3 released
      {
        arrayindex = 7;          // Index to S3 released code
        transmit();              // Transmit S3 released code
      }
      else  // S3 release was not validated after debounce
            // so set up to look only for S3 release
      {
        P2IES &= ~BBIT3;         // Select rising edge on P2.3
        P2IE = BBIT3;            // Enable int request on P2.3 only
        port2flag = 0;           // Clear Port2 flag
        _EINT();                 // Enable interrupts
        _BIS_SR(LPM3_bits);      // Enter LPM3 sleep
        while(port2flag == 0);   // Wait until interrupt
        if(port2flag == 3)       // Interrupt from P2.3
        {
          debouncedelay();
          if(P2IN & BBIT3)       // S3 released
          {
            arrayindex = 7;      // Index to S3 released code
```

```
            transmit();         // Transmit S3 released code
          }
        }
      }
    }
    else  // S3 not yet released, set up and look for it
    {
      P2IES &= ~BBIT3;          // Select rising edge on P2.3
      P2IE = BBIT3;             // Enable int request on P2.3 only
      port2flag = 0;            // Clear Port2 flag
      _EINT();                  // Enable interrupts
      _BIS_SR(LPM3_bits);       // Enter LPM3 sleep
      while(port2flag == 0);    // Wait until interrupt
      if(port2flag == 3)        // Interrupt from P2.3
      {
        debouncedelay();
        if(P2IN & BBIT3)      // S3 released
        {
          arrayindex = 7;     // Index to S3 released code
          transmit();         // Transmit S3 released code
        }
      }
    }
  }
}
if(port2flag == 4)              // Is S4 pressed?
{
  debouncedelay();
  if(!(P2IN & BBIT4))           // Validate S4 pressed
  {
    arrayindex = 3;             // Index to S4 pressed code
    transmit();                 // Transmit S4 pressed code
    if(P2IN & BBIT4)            // Is S4 released?
    {
      debouncedelay();
      if(P2IN & BBIT4)          // Validate S4 released
      {
        arrayindex = 8;         // Index to S4 released code
        transmit();             // Transmit S4 released code
      }
      else  // S4 release was not validated after debounce
            // so set up to look only for S4 release
      {
        P2IES &= ~BBIT4;        // Select rising edge on P2.4
        P2IE = BBIT4;           // Enable int request on P2.4 only
        port2flag = 0;          // Clear Port2 flag
        _EINT();                // Enable interrupts
        _BIS_SR(LPM3_bits);     // Enter LPM3 sleep
        while(port2flag == 0);  // Wait until interrupt
        if(port2flag == 4)      // Interrupt from P2.4
        {
          debouncedelay();
          if(P2IN & BBIT4)      // S4 released
          {
            arrayindex = 8;     // Index to S4 released code
            transmit();         // Transmit S4 released code
          }
        }
      }
    }
    else  // S4 not yet released, set up and look for it
    {
      P2IES &= ~BBIT4;          // Select rising edge on P2.4
      P2IE = BBIT4;             // Enable int request on P2.4 only
      port2flag = 0;            // Clear Port2 flag
      _EINT();                  // Enable interrupts
      _BIS_SR(LPM3_bits);       // Enter LPM3 sleep
      while(port2flag == 0);    // Wait until interrupt
      if(port2flag == 4)        // Interrupt from P2.4
      {
        debouncedelay();
        if(P2IN & BBIT4)      // S4 released
        {
          arrayindex = 8;     // Index to S4 released code
          transmit();         // Transmit S4 released code
        }
```

```
          }
        }
      }
    }
    if(port2flag == 5)              // Is S5 pressed?
    {
      debouncedelay();
      if(!(P2IN & BBIT5))           // Validate S5 pressed
      {
        arrayindex = 4;             // Index to S5 pressed code
        transmit();                 // Transmit S5 pressed code
        if(P2IN & BBIT5)            // Is S5 released?
        {
          debouncedelay();
          if(P2IN & BBIT5)          // Validate S5 released
          {
            arrayindex = 9;         // Index to S5 released code
            transmit();             // Transmit S5 released code
          }
          else  // S5 release was not validated after debounce
                // so set up to look only for S5 release
          {
            P2IES &= ~BBIT5;        // Select rising edge on P2.5
            P2IE = BBIT5;           // Enable int request on P2.5 only
            port2flag = 0;          // Clear Port2 flag
            _EINT();                // Enable interrupts
            _BIS_SR(LPM3_bits);     // Enter LPM3 sleep
            while(port2flag == 0);  // Wait until interrupt
            if(port2flag == 5)      // Interrupt from P2.5
            {
              debouncedelay();
              if(P2IN & BBIT5)      // S5 released
              {
                arrayindex = 9;     // Index to S5 released code
                transmit();         // Transmit S5 released code
              }
            }
          }
        }
        else  // S5 not yet released, set up and look for it
        {
          P2IES &= ~BBIT5;          // Select rising edge on P2.5
          P2IE = BBIT5;             // Enable int request on P2.5 only
          port2flag = 0;            // Clear Port2 flag
          _EINT();                  // Enable interrupts
          _BIS_SR(LPM3_bits);       // Enter LPM3 sleep
          while(port2flag == 0);    // Wait until interrupt
          if(port2flag == 5)        // Interrupt from P2.5
          {
            debouncedelay();
            if(P2IN & BBIT5)      // S5 released
            {
              arrayindex = 9;     // Index to S5 released code
              transmit();         // Transmit S5 released code
            }
          }
        }
      }
    }
  }
}


//==============================================================================
//    Function:   void irtest(void)
//    Purpose:    Test function for debugging IR transmission/reception
//    Algorithm:  Infinite loop transmitting each button code (in the array
//                buttoncodes[])one at a time followed by a 1/2 sec delay.
//                The 10 button codes are sequentially indexed by incrementing
//                the global variable arrayindex up to 9 and then resetting it
//                it to 0 in a continuous loop.
//    Inputs:     None
//    Returns:    Does not return ==> Infinite loop
//    Results:    Infinite loop transmitting each of the button codes followed
//                by a 1/2 sec delay.
//    Includes:   Assumes MSP430 header file is included in main routine
```

```
//               ==> ex #include <msp430x11x1.h>
//    Calls:     void transmit(void)
//               void delay500ms(void)
//===============================================================================
void irtest(void)
{
  while(1)                          // Repeat forever
  {
    arrayindex = 0;                 // Set arrayindex to first array element
    while (arrayindex < 10)         // Sequence thru the 10 array elements
    {
      transmit();                   // Transmit button code
      delay500ms();                 // Delay 1/2 sec
      //debouncedelay();              // *Delay 30ms (for debugging)
      arrayindex++;                 // Point to next array element
    }
  }
}


//===============================================================================
//    Function:   void delay500ms(void)
//    Purpose:    500ms (1/2 sec) delay
//    Algorithm:  500ms is timed using TIMER_A with 32.768kHz ACLK input
//                No interrupts are used.  TIMER_A counts up to 0x3FFF (hex)
//                or 16384 decimal = 500ms
//    Inputs:     None
//    Returns:    None
//    Results:    500ms delay
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//===============================================================================
void delay500ms(void)
{
  // Set up TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared

  // CCR0 is loaded with the # of ACLK cycles to count up to
  // 0x3FFF hex = 16384 decimal ==> 500ms at ACLK = 32.768 kHz

  CCR0 = 0x3FFF;

  // Start TIMER_A counting up

  TACTL |= BIT4;

  while(TAR != CCR0);  // Wait until TIMER_A counts up to CCR0

  // Stop TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared
}


//===============================================================================
//    Function:   void debouncedelay(void)
//    Purpose:    Button press debounce delay
//    Algorithm:  30ms is timed using TIMER_A with 32.768kHz ACLK input
//                No interrupts are used.  TIMER_A counts up to 0x03D7 (hex)
//                or 983 decimal = 30.0ms
//    Inputs:     None
//    Returns:    None
//    Results:    30ms delay
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//===============================================================================
void debouncedelay(void)
{
  // Set up TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared

  // CCR0 is loaded with the # of ACLK cycles to count up to
  // 0x03D7 hex = 983 decimal ==> 30.0ms at ACLK = 32.768 kHz
  // Empirical measurements determined the period to be 30.0ms
```

```
  CCR0 = 0x03D7;

  // Start TIMER_A counting up

  TACTL |= BIT4;

  while(TAR != CCR0);  // Wait until TIMER_A counts up to CCR0

  // Stop TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared
}



//================================================================================
//    Function:   void delay2ms(void)
//    Purpose:    2ms delay
//    Algorithm:  2ms is timed using TIMER_A with 32.768kHz ACLK input
//                No interrupts are used.  TIMER_A counts up to 0x0042 (hex)
//                or 66 decimal = 2.0ms (Empirical measurements tuned
//                the count to 0x0040 (hex) for 2.01ms period)
//    Inputs:     None
//    Returns:    None
//    Results:    2ms delay
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//================================================================================
void delay2ms(void)
{
  // Set up TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared

  // CCR0 is loaded with the # of ACLK cycles to count up to
  // 0x0042 hex = 66 decimal ==> 2.01ms at ACLK = 32.768 kHz
  // 0x0040 hex was found to produce 2.01ms period when measured

  CCR0 = 0x0040;    // Empirically set for 2.01ms

  // Start TIMER_A counting up

  TACTL |= BIT4;

  while(TAR != CCR0);  // Wait until TIMER_A counts up to CCR0

  // Stop TIMER_A

   TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared
}


//================================================================================
//    Function:   void delay1ms(void)
//    Purpose:    1ms delay
//    Algorithm:  1ms is timed using TIMER_A with 32.768kHz ACLK input
//                No interrupts are used.  TIMER_A counts up to 0x0021 (hex)
//                or 33 decimal = 1.0ms (Empirical measurements tuned
//                the count to 0x001F (hex) for 1.00ms period)
//    Inputs:     None
//    Returns:    None
//    Results:    1ms delay
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//================================================================================
void delay1ms(void)
{
  // Set up TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared

  // CCR0 is loaded with the # of ACLK cycles to count up to
  // 0x0021 hex = 33 decimal ==> 1.00ms at ACLK = 32.768 kHz
  // (0x001F hex was found to produce 1.00ms period when measured)
```

```
  CCR0 = 0x001F;    // Empirically set for 1.00ms

  // Start TIMER_A counting up

  TACTL |= BIT4;

  while(TAR != CCR0);  // Wait until TIMER_A counts up to CCR0

  // Stop TIMER_A

   TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared
}


//================================================================================
//    Function:   void delay1point5ms(void)
//    Purpose:    1.5ms delay
//    Algorithm:  1.5ms is timed using TIMER_A with 32.768kHz ACLK input
//                No interrupts are used.  TIMER_A counts up to 0x0031 (hex)
//                or 49 decimal = 1.5ms (Empirical measurements tuned
//                the count to 0x002F (hex) for 1.50ms period)
//    Inputs:     None
//    Returns:    None
//    Results:    1.5ms delay
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//================================================================================
void delay1point5ms(void)
{
  // Set up TIMER_A

  TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared

  // CCR0 is loaded with the # of ACLK cycles to count up to
  // 0x0031 hex = 49 decimal ==> 1.50ms at ACLK = 32.768 kHz
  // 0x002F hex was found to produce 1.50ms period when measured

  CCR0 = 0x002F;    // Empirically set for 1.50ms

  // Start TIMER_A counting up

  TACTL |= BIT4;

  while(TAR != CCR0);  // Wait until TIMER_A counts up to CCR0

  // Stop TIMER_A

   TACTL = 0x0104;   // Clock source = ACLK/1, timer is stopped and cleared
}


//================================================================================
//    Function:   void transmitstart(void)
//    Purpose:    Transmits the start sentinel followed by a space
//    Algorithm:  ACLK (32.768 kHz) is output on P2.0 for 2ms followed by
//                a 1ms space of P2.0 low.  2ms and 1ms timing is done by
//                calling two functions: delay2ms() and delay1ms().  Both
//                functions are assumed to be in the application.
//    Inputs:     None
//    Returns:    None
//    Results:    Transmits the start sentinel followed by a space on P2.0
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//    Calls:      void delay2ms(void)
//                void delay1ms(void)
//================================================================================
void transmitstart(void)
{
  P2SEL |= BBIT0;       // P2.0 = ACLK

  // *For debugging, P2.0 outputs a logic level rather than an ACLK burst
  //P2OUT |= BBIT0;       // *P2.0 set high (for debugging)

  delay2ms();           // Transmit start sentinel for 2ms
```

```
  P2SEL &= ~BBIT0;      // P2.0 reset as output, not ACLK

  //P2OUT &= ~BBIT0;      // *P2.0 set low (for debugging)

  delay1ms();           // Transmit space for 1ms
}


//================================================================================
//    Function:   void transmitone(void)
//    Purpose:    Transmits one followed by a space
//    Algorithm:  ACLK (32.768 kHz) is output on P2.0 for 1.5ms followed by
//                a 1ms space of P2.0 low.  1.5ms and 1ms timing is done by
//                calling two functions: delay1point5ms() and delay1ms().  Both
//                functions are assumed to be in the application.
//    Inputs:     None
//    Returns:    None
//    Results:    Transmits a one followed by a space on P2.0
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//    Calls:      void delay1point5ms(void)
//                void delay1ms(void)
//================================================================================
void transmitone(void)
{
  P2SEL |= BBIT0;       // P2.0 = ACLK

  // *For debugging, P2.0 outputs a logic level rather than an ACLK burst
  //P2OUT |= BBIT0;       // *P2.0 set high (for debugging)

  delay1point5ms();     // Transmit for 1.5ms (one)

  P2SEL &= ~BBIT0;      // P2.0 reset as output, not ACLK

  //P2OUT &= ~BBIT0;      // *P2.0 set low (for debugging)

  delay1ms();           // Transmit space for 1ms
}


//================================================================================
//    Function:   void transmitzero(void)
//    Purpose:    Transmits zero followed by a space
//    Algorithm:  ACLK (32.768 kHz) is output on P2.0 for 1ms followed by
//                a 1ms space of P2.0 low.  1ms timing is done by
//                calling function: delay1ms().  The function is assumed
//                to be in the application.
//    Inputs:     None
//    Returns:    None
//    Results:    Transmits zero followed by a space on P2.0
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//    Calls:      void delay1ms(void)
//================================================================================
void transmitzero(void)
{
  P2SEL |= BBIT0;       // P2.0 = ACLK

  // *For debugging, P2.0 outputs a logic level rather than an ACLK burst
  //P2OUT |= BBIT0;       // *P2.0 set high (for debugging)

  delay1ms();           // Transmit for 1ms (zero)

  P2SEL &= ~BBIT0;      // P2.0 reset as output, not ACLK

  //P2OUT &= ~BBIT0;      // *P2.0 set low (for debugging)

  delay1ms();           // Transmit space for 1ms
}


//================================================================================
//    Function:   void transmit(void)
//    Purpose:    Transmits a button press or release code
//    Algorithm:  The function copies the code pointed to by the variable ==>
//                arrayindex (declared in main()).  arrayindex is used to
```

```
//                point to the button codes in the array ==> buttoncodes[].
//                Since there are only 10 elements in buttoncodes[], valid
//                values for arrayindex are 0 thru 9.  If arrayindex is not
//                a valid value, there is no transmission and the function
//                just returns.  If arrayindex is a valid value, then it is
//                copied into temp.  If temp is equal to zero, there is no
//                transmission and the function just returns.  A zero code
//                indicates that no transmission should occur.  If temp is
//                non zero, then the least significant 8 bits of the code in
//                temp will be transmitted (MSB first).  The function calls
//                transmitstart() to transmit the start sentinel.  Based on
//                the value of bit 7 of temp either transmitone() or
//                transmitzero() is called.  Each of these functions transmits
//                its 32.768 kHz burst on P2.0 followed by a space.  temp is
//                then bit shifted left one bit and the process repeats.  After
//                all 8-bits have been transmitted, the function returns.
//                The PCM protocol used is a 32.768 kHz IR burst for 2ms as a
//                start sentinel with a 1.5ms burst as a "1" and a 1ms burst
//                as a "0".  There is a 1ms break (with no IR transmission)
//                following each IR burst.  The output is on P2.0 where it
//                outputs ACLK (32.768 kHz) for the burst and outputs a low
//                for the spaces.
//
//                    _||||__||__|||__||__||__|||__||__|||__|||__
//                      "S" "0"  "1"  "0" "0" "1" "0"  "1"  "1"
//                          "S" = start sentinel,  "__" = space,
//                          "||||, |||, ||" = IR burst transmission
//
//                NOTE: Since the button codes in buttoncodes[] are 16-bit
//                and only the lower 8 bits are actually transmitted, the
//                unused upper MSB 8 bits should be all zeros in the case
//                where no transmission is desired.  (For example, 0100
//                (hex) will result in the transmission of 00000000 ==> 8
//                zero's rather no transmission at all like 0000 (hex).)
//                For any other codes, the upper MSB 8 bits of the 16-bit
//                code word are "don't care" values.
//    Inputs:     None
//    Returns:    None
//    Results:    Transmits button press/release code on P2.0
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//                Main routine needs to have array buttoncodes[] with
//                10 unsigned int values and unsigned int arrayindex.
//    Calls:      void transmitstart(void)
//                void transmitone(void)
//                void transmitzero(void)
//==============================================================================
void transmit(void)
{
  unsigned int temp;
  unsigned int i;

  if(arrayindex < 10)  // Check if index is in the valid range 0-9
  {
    temp = buttoncodes[arrayindex];       // Copy the button code

    if(temp!= 0)                          // Only transmit if code is non zero
    {
      transmitstart();                    // Transmit start sentinel

      for(i=8; i!=0; i--)
      { // Only the least significant 8 bits of the button code in temp
        // are transmitted.

        if(temp & BIT7)                   // Is bit 7 a "1"?
        {
          transmitone();                  // Yes, transmit a "1"
        }
        else
        {
          transmitzero();                 // No, transmit a "0"
        }
        temp <<= 1;                       // Bit shift temp left one bit
      }
    }
  }
```

```
}


//==============================================================================
//    Function:   interrupt[PORT2_VECTOR] void port2isr(void)
//    Purpose:    Port_2 Interrupt Service Routine
//    Algorithm:  Disables interrupts and then determines which of P2.1-P2.5
//                inputs caused the Port2 interrupt.  A global variable,
//                port2flag, is set to a value from 1 to 5 corresponding to
//                the input causing the interrupt.  If interrupts have occurred
//                on multiple inputs, then only the lowest (P2.5 highest to P2.1
//                lowest) interrupt is saved.  Only if port2flag = 0, is the
//                system prepared to accept new Port2 interrupts.  The routine
//                then clears P2IFG and wakes up the system from LPM3 sleep
//                using intrinsic _BIC_SR_IRQ(LPM3_bits).
//
//                NOTE: This ISR disables all interrupts.  Interrupts must be
//                enabled again in the main application.
//
//    Includes:   Assumes MSP430 header file is included in main routine
//                ==> ex #include <msp430x11x1.h>
//==============================================================================
interrupt [PORT2_VECTOR] void port_2isr(void)
{
  _DINT();                  // Disable interrupts

  if(port2flag == 0)        // If the system is not already addressing a
                            // previous button press/release, then determine
                            // which button was responsible for this interrupt
                            // on Port2.  If more than one interrupt occurred
                            // at the same time, only acknowledge the lowest
                            // Port2 input pin (P2.5 highest to P2.1 lowest)
                            // involved.
  {
    if(P2IFG & BBIT5)
    {
      port2flag = 5;        // S5 caused interrupt
    }
    if(P2IFG & BBIT4)
    {
      port2flag = 4;        // S4 caused interrupt
    }
    if(P2IFG & BBIT3)
    {
      port2flag = 3;        // S3 caused interrupt
    }
    if(P2IFG & BBIT2)
    {
      port2flag = 2;        // S2 caused interrupt
    }
    if(P2IFG & BBIT1)
    {
      port2flag = 1;        // S1 caused interrupt
    }
  }

  P2IFG = 0;                // Clear Port2 interrupt flag

  _BIC_SR_IRQ(LPM3_bits);   // Wake up system from LPM3 sleep
}
```