;------------------------------------------------------------------------

;

; File: tinytemp.asm

;

; Tiny Temperature Domonstration Board Code

;

; written under contract for:

; Analog Devices Inc.

; Wilmington, MA

;

; hardware design and firmware coding by:

; Norm Bernstein

; Marisystems, Inc.

; 77 Tosca Drive

; Stoughton, MA 02072

; (781) 341-3611

; msys@marisystems.com

;

; Start date: 2/20/01

;

;

; DESCRIPTION OF APPLICATION -- this code is used in the Temperature Logger

; demonstration board, which utilizes the ADuC824 microcontroller.

;

; This application operates in two major modes:

;

; Display mode: At power-up, the application operates in the display mode.

; In this mode, temperature measurements are made twice per second, and

; displayed on the LCD display. The data is also transmitted via the serial

; port, and is logged to data FLASH every five minutes. The pushbutton may

; be used to switch between celsius and fahrenheit. A dumb terminal, or the

; companion GUI application, may be used to access the application's data, and

; control such things as the selection of sensor units as well as to

; download the logged data within the data FLASH memory.

;

; Data-logging mode: After thirty minutes, the application switches to the

; datalogging mode. In this mode, the application goes into a low power mode,

; with the display showing 'Log'. Every ten minutes, the application 'wakes

; up' long enough to take a single temperature measurement and store it to

; data FLASH within the ADuC824. Using 512 bytes of data FLASH, the logging

; will continue for 23+ hours. Once the 512 bytes allotted has been filled,

; logging continues, overwriting the oldest data (It's a circular buffer).

;

; (NOTE: the aforementioned thirty minute timer is reset

; whenever the C/F button is pressed, and/or whenever serial comms have been

; received from either a dumb terminal or the GUI application.)

;

;---------------------------------------------------------------------

;

; NOTE on power conservation: In order to extend battery life as long

; as possible, this application will employ the 'power down' mode to

; the maximum extent possible. The long interval timer function will be

; employed as the main pacing mechanism. During the display and GUI modes,

; the pacer wil lbe set up to generate a 'wake up' every 0.5 seconds;

; during the datalogging mode, the interval will be extended to five

; minutes.

;

;---------------------------------------------------------------------

;

; ASSEMBLER DIRECTIVES

;

;---------------------------------------------------------------------

;

$NOMOD51 ;rig for 824 definitions

$INCLUDE (ADuC824.INC)

;

;-------------------------------------------------------------------------

;

; EXTERNAL DECLARATIONS -- this section contains all declarations of

; external variables and routines which are contained in another

; module (file)

;

;-------------------------------------------------------------------------

;

; (none used)

;

;-------------------------------------------------------------------------

;

; INTERNAL DECLARATIONS -- this section contain declarations internal

; to this file, or relating to registers or device controls used in this

; application. These declarations are usually made for convenience and

; clarity

;

;-------------------------------------------------------------------------

;

FCHAR EQU 0F4H ;seven segment byte for the 'F' character

CCHAR EQU 0BCH ;seven segment byte for the 'C' character

;

;-------------------------------------------------------------------------

;

; PORT DECLARATIONS AND ASSIGNMENTS -- this section defines the usage of

; the port bits

;

;-------------------------------------------------------------------------

;

PBSWITCH EQU P3.2 ;pushbutton switch

;

;-------------------------------------------------------------------------

;

; SYSTEM CONSIDERATIONS

;

;----------

;

; Clock -- this application is designed for extreme low power

; consumption. Accordingly, we will be using the default core

; clock rate of 1.572864 MHz, which provides for an instruction

; cycle period of 7.629394 uS.

;

;----------

;

; A/D scaling -- this application accesses two possible sensors; an

; RTD, or the onboard chip temperature sensor.

;

; The RTD is rigged for ratiometric operation, with respect to an

; external precision resistance (12.4K). The RTD is in series with the

; resistance, and the string is excited by one of the onboard current

; sources, which runs at approximately 200 uA. The voltage across the

; precision reference resistor feeds the reference input of the ADuC824.

;

; At 200uA, the drop across the reference resistor is approximately

; 2.48 volts, which is considered optimum for the ADuC824. The RTD is

; nominally 1000 ohms @25C, but it rises to approximately 1328 ohms

; at +85C, the maximum operating temperature for this application.

; Consequently, the maximum voltage drop across the RTD will be on the

; order of 265.6 millivolts. If we choose the 320 mV input gain range,

; the resulting conversion output will have a scale of:

;

; D(out) = (R(rtd)/12.4K) \* (2.5/0.320)

;

; and will be invariant with the magnitude or drift of the excitation

; current.

;

; The chip temperature sensor can only be measured by the auxilliary

; ADC, and has a nominal output of 0 for 0 degrees C, with a slope of

; 256 LSB's (out of a 16 bit conversion) per degree C. This scaling is

; valid when using the internal reference voltage.

;

;----------

;

; A/D conversion period -- in this application, we are selecting the

; fastest standard speed for conversion, to yield the lowest power..

; Using an SF coefficient of 0DH, the conversion time is 9.52ms. However,

; since we'll be using the 'single conversion mode', the actual time from

; initiation of a conversion, to completion as signalled by the RDY bit,

; is twice that period, or 19.04 mS.

;

;-------------------------------------------------------------------------

;

;

;

;-------------------------------------------------------------------------

;

; MACRO DEFINITIONS -- this section contains the definitions of any

; macros which may be used within this program

;

;-------------------------------------------------------------------------

;

; (no macros used)

;

;-------------------------------------------------------------------------

;

; TIMER USAGE -- the 8052 functional model has three timer/counters.

;

;-------------------------------------------------------------------------

;

; TIMER2 will be used for baud rate generation

;

;-------------------------------------------------------------------------

;

; EEPROM ALIASES -- aliases for the bytes within a data FLASH page

; (4 bytes)... these make acessing frequently used data FLASH variables

; somewhat easier

;

;-------------------------------------------------------------------------

;

; Aliases used for the LOGCONTROL page of data FLASH

;

LOGPTR EQU EDATA1 ;pointer to the next available byte pair of data

;FLASH memory

;

;-------------------------------------------------------------------------

;

; EEPROM ALLOCATIONS -- this section contains the allocations of the

; ADuC812's EEPROM (really, FLASH DATA memory). The values of these

; variables refer to the 'page' of FLASH memory.

;

;-------------------------------------------------------------------------

;

LOGBUF EQU 0 ;pages 0 through 63 constitute the buffer

;used for logging temperature while the

;application is in the logging mode

LOGCONTROL EQU 128 ;this page contains control bytes for logging.

;

FIRSTRUN EQU 129 ;this page is used for virgin board detection.

;The first time the application is run, we want

;to fill data FLASH with a known pattern.

;

;-------------------------------------------------------------------------

;

; RAM ALLOCATIONS -- this section defines the allocation of the

; processor's RAM resources

;

;-------------------------------------------------------------------------

;

DSEG

;

;---------

;

; Register bank space occupies 00H through 1FH, organized as four

; banks of eight bytes. All 8051-architecture designs will use bank

; 0 (00H through 07H), but apps which do not require context switching

; can use the space occupied by the other three banks as ordinary

; directly addressable RAM

;

;

;----------

;

DSEG AT 18H

ORG 18H

;

;----------

;

; Direct bit addressable registers: these start at absolute address

; 20H and continue to address 2FH

;

DSEG AT 20H

ORG 20H

;

;

GPBITS: DS 1 ;general purpose booleans

SGNBIT EQU GPBITS.0 ;sign of the output value (1=negative, 0=positive)

CFFLG EQU GPBITS.1 ;0=celsius, 1=fahrenheit

SNSRFLG EQU GPBITS.2 ;0=chiptemp, 1=RTD

CMDFLG EQU GPBITS.3 ;bit signals that an incoming command awaits

TXRDY EQU GPBITS.4 ;semaphore for the TI bit in serial comms

LOGSRC EQU GPBITS.5 ;temporary for logged data source

LOGUNIT EQU GPBITS.7 ;temporary for logged data units

;

GPBITS2: DS 1 ;additional general purpose booleans

OVERFLOW EQU GPBITS2.0 ;Overflow out of the ADCLIN routine

UNDERFLOW EQU GPBITS2.1 ;underflow out of the ADCLIN routine

LOGMODE EQU GPBITS2.2 ;indicates we're in the logging mode

CNVREADY EQU GPBITS2.3 ;semaphore signals that a conversion is ready

RTDEXISTS EQU GPBITS2.4 ;set if RTD exists in the hardware

SGNBITX EQU GPBITS2.5 ;sign bit used for intermediate calcs when multiplying

SGNBITY EQU GPBITS2.6 ;sign bit used for intermediate calcs when multiplying

LOGFLG EQU GPBITS2.7 ;flag to indicate it's time to log data

;

GPBITS3: DS 1 ;additional general purpose bits

TEMPCFFLG EQU GPBITS3.0 ;temporary copy of CFFLG for computing logged data

TEMPSNSRFLG EQU GPBITS3.1 ;temporary copy of SNSRFLG for xmitting logged data

DWNLDFLG EQU GPBITS3.2

;

;

TEMP: DS 2 ;temperature variable. We establish this in bit-addressable

;memory to make access of the sign bit easier

;

DNLDNDX: DS 1 ;an index for downloading logged data

DNLDCTR: DS 1 ;a counter for downloading logged data

LOGDATHI: DS 1 ;temporary storage for logged data

LOGDATLO: DS 1

;

;----------

;

; Regular directly addressable RAM resumes at address 30H and continues

; through address 7FH

;

DSEG AT 30H

ORG 30H

;

;

; Definitions for the LCD display buffer -- this buffer contains the 31 bits

; which will be shifted out to the LCD controller chip.

;

LCDMSB: DS 1 ;7 LSB's contain segments a-g of leftmost full digit, msb drives

;the vertical portion of the 'plus' sign (actually, the leftmost

;colon, and not used in this application)

;

LCD2SB: DS 1 ;7 LSB's contain segments a-g of next digit, msb drives the 1's

;digit on the left side

;

LCD3SB: DS 1 ;7 LSB's contain segments a-g of next digit, msb drives the

;horizontal portion of the 'plus' sign (actually, the minus

;sign)

;

LCDLSB: DS 1 ;7 LSB's contain segments a-g of rightmost digit, msb drives

;nominal decimal point (2nd from right), NOTE: segment C not

;actually driven, only 31 bits in the display driver shift reg.

;

;----------

;

PBCDBUF: DS 3 ;buffer for the output of integer-to-bcd conversions

CMDCHAR: DS 1 ;buffer for incoming commands

LOGCTR: DS 1 ;counter to determine when we switch to the logging mode, from

;the gui/display mode

;

;

;----------

;

; Indirectly addressable RAM starts at 80H and grows upwards to 0FFH

; (but must include stack space)

;

ISEG AT 80H

ORG 80H

;

;----------

;

; Stack grows upwards and starts after all declared variables

;

STKSTRT EQU $

;

;

;-------------------------------------------------------------------------

;

; CODE SEGMENT STARTS HERE

;

;-------------------------------------------------------------------------

;

CSEG AT 0

;

SJMP COLD\_INIT ;and go to the cold init

;

;-------------------------------------------------------------------------

;

; INTERRUPT VECTOR SPACE -- interrupt vector space starts at 03H and

; occupies specific addresses through 05BH. Unimplemented interrupts

; should never execute, but we'll give them RETI instruction anyhow,

; just in case of code accidents

;

;-------------------------------------------------------------------------

;

ORG 03H ;pushbutton interrupt

LJMP PBINT

;

ORG 0BH ;TF0 interrupt (unused)

RETI

;

ORG 13H ;IE1 interrupt (unused)

RETI

;

ORG 1BH ;TF1 interrupt (unused)

RETI

;

ORG 23H ;serial port interrupt

LJMP SPINT

;

ORG 2BH ;TF2/EXF2 interrupt (unused)

RETI

;

ORG 33H ;RDY0/RDY1 ADC interrupt (unused)

RETI

;

ORG 3BH ;I2C/SPI interrupt (unused)

RETI

;

ORG 43H ;PSMI interrupt (unused)

RETI

;

ORG 53H ;timer interval counter interrupt

LJMP TICINT

;

ORG 5BH ;watchdog interrupt (unused)

RETI

;

;-------------------------------------------------------------------------

;

; COLD BOOT INITIALIZATION -- the following section constitutes

; the initialization of various registers within the ADuC824, prior

; to entering the execution phase of the program

;

;-------------------------------------------------------------------------

;

; initialize various RAM variables

;

COLD\_INIT:

;

MOV SP,#STKSTRT ;set the stack pointer first

;

;

;-------------

;

; Initialize any I/O ports first

;

MOV SPICON,#03FH ;enable the SPI register for LCD interface

;

;-------------

;

; Set up the A/D's for the appropriate conditions. We'll want the

; 200 uA current source turned on, and the gain and mux set appropriately.

;

MOV ADC0CON,#044H ;external reference, AIN1/2 differential,

;unipolar, gain range 320 mV

MOV ADC1CON,#020H ;internal reference, Temp sensor input,

;bipolar

MOV ICON,#001H ;current source 1 enabled and routed to

;IEXC1 (pin 3), source 2 turned off,

;

;-------------

;

; Write the opening logo on the display, and hold it for two seconds

;

MOV LCDMSB,#077H

MOV LCD2SB,#05BH

MOV LCD3SB,#90H

MOV LCDLSB,#40H

LCALL DSPLSHIFT

;

MOV R1,#2

INIT\_05:

MOV R2,#0

INIT\_10:

MOV R3,#0

DJNZ R3,$ ;inner loop, 3.722 mS

DJNZ R2,INIT\_10 ;middle loop, 256 executions ~950 mS

DJNZ R1,INIT\_05 ;outer loop, 2 executions ` 1.9 sec

;

;-------------

;

; Clear all RAM, excluding the lowest 2 register banks, first. There's

; no specific requirement to do this, except that it's very helpful in

; debugging, because it shows clearly, in simulation or emulation, what

; RAM locations have been initialized, and what the extent of the stack

; is. One useful side effect of clearing RAM is that we clear all

; variables automatically...

;

MOV R0,#10H ;start at 10H (the third register bank)

INIT\_20:

MOV @R0, #0 ;write a zero

INC R0 ;next location

CJNE R0,#0H,INIT\_20 ;repeat until all memory cleared

;

;-----------

;

; If this is the first execution of this board, we need to fill the

; log buffer portion of data FLASH with a default pattern (00FAH, or 25.0 deg).

; This allows us to observe default initial data easily.

;

MOV EADRL, #FIRSTRUN ;let's examine the data in the FIRSTRUN

MOV ECON,#01H ;read the page

MOV A,EDATA1 ;look at the first byte

JZ INIT\_50 ;if zero, we're already initialized

;

; If the first byte of FIRSTRUN is non-zero, it means we have to initialize

; the data FLASH.

;

MOV EDATA1,#000H ;the pattern will be 00FAH (250), which is well

MOV EDATA2,#0FAH ;within the normal data range

MOV EDATA3,#000H

MOV EDATA4,#0FAH

MOV EADRL,#0 ;start at the first page of the circular buffer

MOV R2,#128 ;128 pages to flush and fill

;

INIT\_30:

MOV ECON,#05H ;erase the page

MOV ECON,#02H ;write the page

INC EADRL ;next page

DJNZ R2,INIT\_30 ;repeat for all 64 pages

;

; We'll initialize the LOGCONTROL page

MOV EADRL,#LOGCONTROL

MOV EDATA1,#0

MOV EDATA2,#0

MOV EDATA3,#0

MOV EDATA4,#0

MOV ECON,#05H

MOV ECON,#02H

;

; We'll also write a 0 to the FIRSTRUN page, to indicate that this board has

; been initialized

;

MOV EADRL,#FIRSTRUN

MOV EDATA1,#0

MOV EDATA2,#0

MOV EDATA3,#0

MOV EDATA4,#0

MOV ECON,#05H

MOV ECON,#02H

;

;-----------

;

; Since we'll be making use of the TIC (Timer Interval Counter),

; as well as the power-down mode, we're going to want the 32.788KHz

; oscillator to keep on running in powerdown mode. We will want the

; application to respond quickly to interrupts, so we'll set the

; FINT bit within PLLCON to allow for fast interrupt response.

;

INIT\_50:

MOV PLLCON,#03H+08H ;03H is the default value, 08H

;sets FINT

;

;-----------

;

; During the GUI/Display mode of operation, each interrupt

; will result in a conversion of the input, the setting of the LCD

; display, and the generation of a serial output string. This occurs

; at a 0.5 second period. During the Logging mode, the interval will

; be extended to 5 minutes, and at each interrupt, a single conversion

; will be made, and the data will be logged to data FLASH

;

; Since we'll start out in the Display mode, we'll initialize the TIC

; for 0.5 second operation.

;

MOV TIMECON,#43H ;set for 1/128 second timebase

MOV INTVAL,#40H ;64 iterations of 1/128 sec = 0.5 sec

MOV SEC,#0 ;zero the seconds

MOV MIN,#0 ;and the minutes

;

;-----------

;

; This is an ADuC824 comms initialization. Since we want 9600 baud operation,

; but we're using the default core clock rate of 7.629394 uS, we'll need to

; use TIMER2 for baud rate generation. The following preload values will

; result in a baud rate of 9830, which is close enough to 9600 baud for our

; purposes

;

SETB RCLK ;set the bits to enable TIMER2-based

SETB TCLK ;baud rate generation

;

MOV RCAP2H,#0FFH ;set the appropriate preload value

MOV RCAP2L,#0FBH

MOV TH2,#0FFH

MOV TL2,#0FBH

;

MOV SCON,#50H ;serial port mode 1, RI flag cleared

MOV T2CON,#34H

;

;-----------

;

; We'll be using the A/D in the single conversion mode, with a minimal

; digital filter constant setting. At this setting, the conversion resolution

; and noise will be 15 bits or better, which is more than adequate for our

; purposes. This setting produces a conversion period of 9.52mS, and since

; we're powering up for each conversion, the actual time to first conversion

; will be twice the period, or 19.04mS

;

MOV SF,#0DH ;set for f(adc)=105.3 Hz

;

;-----------

;

; Kickstart the serial port process by setting TXRDY high to indicate that the

; xmit side of the UART is available

;

SETB TXRDY

;

;-----------

;

; Prior to entering the main execution loop of this application, we want

; to test to see if this particular board has an RTD sensor installed; in

; some distributions of this board, the RTD sensor may not be installed.

;

; We need to kick off a single conversion here

;

MOV ADCMODE,#32H ;kick off a single conversion of the

;primary and auxilliary A/D

JNB RDY0,$ ;and wait for completion

;

; If the primary A/D is saturated, it means that there's no RTD or reference

; resistor installed.

;

MOV A,ADC0H ;read the upper byte

CJNE A,#0FFH,INIT\_60 ;if it's not saturated, the RTD exists

CLR RTDEXISTS ;else mark it as non-existent,

CLR SNSRFLG ;and select chip temperature instead

SJMP INIT\_70

;

INIT\_60:

SETB RTDEXISTS ;else set it to indicate there's an RTD

SETB SNSRFLG ;and make it the power-up default

;

;-----------

;

; OK, time to set up the interrupt enables and priority

;

; Set the PBINT to high priority

;

INIT\_70:

MOV IP, #01H ;set PX0 high

;

; Set the PBINT for edge sensitive operation

;

MOV TCON,#01H ;set IT0 high

;

; Enable the serial port interrupt, the TCI interrupt, and the PB interrupt

;

SETB ES ;enable the serial port interrupt

SETB EX0 ;enable the external interrupt

MOV IEIP2,#04H ;enable the TIC interrupt counter

SETB EA ;enable all interrupts

;

;----------

;

; Before entering the main application loop, we'll initialize and

; enable the watchdog timer mechanism. (note: disabled until release, since

; it interferes with emulation)

;

; MOV WDCON,#41H ;enable the watchdog timer for 64 mS timeout

;

;-------------------------------------------------------------------------

;

; MAIN LOOP -- The MAIN loop constitutes the main portion of the

; application. This application has no specific idle loop task, per se;

; instead, it spends the majority if it's time in either the idle mode

; or powerdown mode, waiting for an interrupt to prompt action.

;

; Because of the mechanization of the power-down and idle modes, we

; need to initiate these modes from a point of main program flow, and

; NOT from within an interrupt. Therefore, we'll use a 'main loop'

; structure which initiates the idle or powerdown mode, and then deals

; the functions required by the interupts once they occur. Signalling

; will be achieved with semaphores.

;

;-------------------------------------------------------------------------

;

MAIN\_LOOP:

;

; I always reinitialize the stack within the main loop of any appli-

; cation. In theory, if there are no stack misalignment errors, this

; isn't necessary. However, I've found that sometimes, doing this will

; correct an undetected stack misalignment error which has creeped

; in... and it doesn't really interfere with debugging. If there are no

; undetected stack misalignments, doing this doesn't hurt. If there ARE

; undetected stack misalignments, resetting the stack here may be enough

; to resolve the problem and at least keep us running.

;

MOV SP,#STKSTRT ;set the stack pointer first

;

;------------- Drop to Power-down or Idle mode -------------

;

; It's time to either go into the powerdown or idle modes. We'll use

; the power-down mode while logging, for maximum power conservation;

; in the logging mode, the only routinely expected interrupt will be

; the TIC (long interval counter) interrupt, although we may also get

; woken up by a depression of the C/F select button.

;

; In the GUI or display modes, we'll use the idle mode, so that the

; serial port interrupt remains active, and so that a user with a

; terminal emulation program can still have access between samples.

;

;

JB LOGMODE,MAIN\_LOOP\_10 ;are we in the logging mode?

;

; We're not in the logging mode, so we'll prepare to go into the

; idle mode

;

MOV A,#31H ;INT0 powerdown interrupt enabled

SJMP MAIN\_LOOP\_15 ;idle mode bit set

;

; We'll vector here if we're in the logging mode, so we'll use

; the power-down mode

;

MAIN\_LOOP\_10:

MOV A,#32H ;power-down mode bit set

MAIN\_LOOP\_15:

MOV PCON,A ;enter idle or powerdown

;

;------------

;

; We're now in the power-down or idle mode!

;

;------------- Return from Power-down or Idle Mode via Interrupt -------

;

; Here is where we end up once an interrupt has taken us out of

; a power-down or idle mode. All we really need to do is to examine

; and process any active semaphores before dropping back into the

; idle or power-down modes by jumping back to the top of the main

; loop.

;

;-----------------------------------------------------------------------

;

JNB CNVREADY,MAIN\_LOOP\_90 ;is a conversion ready?

;

; OK, a conversion has just completed, so we can deal with it

;

CLR CNVREADY ;clear the semaphore for the next pass

;

;-------------------

;

; Before we deal with displaying data, we'll need to see about

; whether it's time to log or not.If the RTD exists, it's the

; one we'll log, regardless of which sensor is selected for display

;

JNB RTDEXISTS,MAIN\_LOOP\_17 ;if it doesn't exist, then we'll

;log chip temp instead

;

; If LOGFLG is set, it's time to log the data.

;

JNB LOGFLG,MAIN\_LOOP\_20 ;skip if not time to log

;

; We're going to log this data... first re-compute it for celsius

;

MOV TEMP,ADC0H ;xfer the A/D to the TEMP register pair

MOV TEMP+1,ADC0M

CPL TEMP.7 ;complement MSB to form two's comp binary

MOV DPTR,#CLINTAB ;logged data is always C

LCALL ADCLIN

CLR LOGFLG ;ok, clear the flag

LCALL LOGGIT ;and log the data

SJMP MAIN\_LOOP\_20 ;we can now deal with displayed data

;

;--------------------

;

; The RTD doesn't exist, so we'll log the chip temp data, presuming

; the LOGFLG is set.

;

MAIN\_LOOP\_17:

JNB LOGFLG,MAIN\_LOOP\_20 ;skip if not time to log

;

MOV TEMP, ADC1H ;fetch the high byte

MOV TEMP+1,ADC1L ;and the low byte

CPL TEMP.7 ;convert to two's comp

CLR TEMPCFFLG ;always log celsius

LCALL CHIPTEMPSCALE

CLR LOGFLG ;ok, clear the flag

LCALL LOGGIT ;and log the data

;

;

;-------------------- RTD display data processing -----------------

;

; If the RTD exists, and the sensor control flag indicates that the user

; has selected the RTD sensor, we'll process it's data as the default for

; display and logging

;

MAIN\_LOOP\_20:

;

JB LOGMODE,MAIN\_LOOP\_90 ;if we're in the log mode, we won't display data

JNB RTDEXISTS,MAIN\_LOOP\_50 ;if it doesn't exist, then we'll skip;

JNB SNSRFLG,MAIN\_LOOP\_50 ;if chip temp is selected, do that one

MOV TEMP,ADC0H ;transfer the A/D to the TEMP register pair

MOV TEMP+1,ADC0M

CPL TEMP.7 ;complement MSB to form two's comp binary

;

; Determine what units are selected

;

JB CFFLG,MAIN\_LOOP\_30 ;jump if it's fahrenheit

MOV DPTR,#CLINTAB ;it's celsius, so rig for celsius lin

SJMP MAIN\_LOOP\_40

MAIN\_LOOP\_30:

MOV DPTR,#FLINTAB ;it's fahrenheit, so rig for fahrenheit lin

MAIN\_LOOP\_40:

LCALL ADCLIN ;linearize and scale the conversion

SJMP MAIN\_LOOP\_60

;

;-------------------- chip temperature data processing --------------

;

; If the RTD does NOT exist, or if the user has intentionally selected the chip

; temperature, we'll process that data instead

;

MAIN\_LOOP\_50:

MOV TEMP, ADC1H ;fetch the high byte

MOV TEMP+1,ADC1L ;and the low byte

CPL TEMP.7

MOV C,CFFLG ;grab the units flag

MOV TEMPCFFLG,C ;and save it for scaling purposes

LCALL CHIPTEMPSCALE

;

;-------------------- display mode processing -------------

;

; We can now deal with the scaled and/or linearized data. We'll convert it,

; display it on the LCD, and transmit it out the serial port

;

MAIN\_LOOP\_60:

JB LOGMODE,MAIN\_LOOP\_90 ;no display or serial data update in log mode

LCALL TEMP\_CONVERT ;convert the temperature to packed BCD

LCALL DISPLAY\_FORMAT ;format and load the data to the LCD display

LCALL LIVEDATAOUT ;generate the serial output datastream

;

;-------------------- serial command processing -----------

;

; Now we'll check to see if there's a serial port command waiting; if there

; is, we'll service it.

;

MAIN\_LOOP\_90:

JNB CMDFLG,MAIN\_LOOP\_95 ;no command waiting? Bypass the service

;

; By definition, if we've received a comand via the serial port, we're supposed

; to return to the display/GUI mode. If we've been in the logging mode, we'll

; need to alter the TIC interval back to 0.5 sec

;

JNB LOGMODE,MAIN\_LOOP\_93 ;if we're already in display mode, no need

;to alter the TIC interval

;

; OK, we've just been awoken by a serial port interrupt, so it's time to

; alter the TIC interval back to 0.5 seconds

;

MOV TIMECON,#43H ;set for 1/128 second timebase

MOV INTVAL,#40H ;64 iterations of 1/128 sec = 0.5 sec

;

MAIN\_LOOP\_93:

LCALL CMD\_SERVICE

;

;

;--------------------------

;

MAIN\_LOOP\_95:

LJMP MAIN\_LOOP ;repeat the loop forever

;

;----------------------------------------------------------

;

; LOGGIT -- logs the latest data point into the curcular data FLASH

; buffer. The data is sitting in TEMP|TEMP+1

;

;----------------------------------------------------------

;

LOGGIT:

;

PUSH TEMP ;save the data

PUSH TEMP+1 ;will be needed again for display

;

; First off, we'll embed the sensor bit in the data field

; before storing it

;

MOV C,SNSRFLG ;grab the sensor flag

MOV TEMP.7,C ;and embed it

;

; We need to figure out where this data is going. To do that, we'll fetch

; the LOGCONTROL page, and examine LOGPTR

;

MOV EADRL,#LOGCONTROL ;fetch the page

MOV ECON, #01H ;and read the page

MOV A,LOGPTR ;fetch the log pointer

INC LOGPTR ;and increment it

MOV ECON,#05H ;erase the page

MOV ECON,#02H ;and write the page

;

; If the pointer is odd, the data drops into the EDATA1|EDATA2 pair. If

; it's even, it drops into the EDATA3|EDATA4 pair

;

PUSH ACC ;save the pointer for now

RR A ;shift to form a page index

ANL A,#07FH ;mask for 7 bits

MOV EADRL,A ;set the address

MOV ECON,#01H ;read the page

;

; Now we've got the page, so we'll put the data where it ought to go

;

POP ACC ;get the pointer back

JB ACC.0,LOGGIT\_10 ;if lsb=1, it's the upper bytes

;

MOV EDATA1,TEMP ;load the data to the first pair

MOV EDATA2,TEMP+1

SJMP LOGGIT\_20

LOGGIT\_10:

MOV EDATA3,TEMP ;load the data to the second pair

MOV EDATA4,TEMP+1

;

; Now we can erase and re-write the page

;

LOGGIT\_20:

MOV ECON,#05H ;erase the page

MOV ECON,#02H ;and write the page

;

POP TEMP+1 ;restore the data

POP TEMP

RET

;

;---------------------------------------------------------------------------

;

; PUSHBUTTON INTERRUPT -- this interrupt is generated by a depression of

; the pushbutton (connected to INT0), and has two possible effects:

;

; 1) if the board is up and running in the display/GUI mode, the pushbutton

; will toggle the board between celsius and fahrenheit. The routine is

; rigged so that units are only changed if the button is held for at

; least two seconds

; 2) if the board is in the logging mode, the pushbutton will 'wake up' the

; application and put it back into the display/GUI mode

;

;---------------------------------------------------------------------------

;

PBINT:

PUSH PSW ;save program status

SETB RS0 ;select the second register bank

;

MOV LOGCTR,#0 ;any PB int resets the timer for switching to log

;

; Were we in the logging mode? If so, it's time to switch back to the

; display/GUI mode

;

JNB LOGMODE,PB\_10 ;skip this if not in the logging mode

;

;------------------

;

; We were in the logging mode when we got here

;

CLR LOGMODE ;disable the logging mode

;

; OK, we've just been awoken by a PB interrupt, so it's time to

; alter the TIC interval back to 0.5 seconds

;

MOV TIMECON,#43H ;set for 1/128 second timebase

MOV INTVAL,#40H ;64 iterations of 1/128 sec = 0.5 sec

;

; Also, reset the MIN counter, the SEC counter, and the LOGCTR so we properly

; restart the timeout process

;

MOV MIN,#0

MOV SEC,#0

MOV LOGCTR,#0

;

SJMP PB\_90 ;skip the units toggle section,

;

;------------------

;

; This is not a wake-up, so we'll do the timeout test to see if the guy REALLY

; wants to switch units. First, we wait for 20 milliseconds to insure that we're

; past the 'bounce'

;

PB\_10:

CLR EA ;suspend all other ints while we do this

MOV R2,#5

PB\_12:

MOV R3,#0

DJNZ R3,$ ;inner loop, 3.722 mS

DJNZ R2,PB\_12 ;5 iterations = 18.6 mS

;

; Now we monitor the switch to insure that it stays low for at least 2

; seconds. If it does, we can switch units. If it doesn't, then we'll

; monitor for a continuous high to insure we're past the trailing edge

; bounce

;

MOV R2,#0

MOV R3,#0

PB\_15:

JB PBSWITCH, PB\_50 ;if it should go high, we're aborting

DJNZ R3,PB\_15 ;this inner loop takes 7.8125mS

DJNZ R2,PB\_15 ;this outer loop takes 2 sec

;

; OK, we passed the criteria, so we can toggle the units flag

;

CPL CFFLG

;

; If we failed the two second criteria, we simply wait for a logic

; '1' and wait for it to stay there for 20 mS before exiting

;

PB\_50:

;

; Exit testing: we poll for a logic '1', and then poll for continuous

; logic 1 for roughly 20 mS before exiting and returning

;

JNB PBSWITCH,$ ;poll for the logic 1

;

PB\_55:

MOV R3,#3

PB\_60:

MOV R2,#0

PB\_65:

JNB PBSWITCH,PB\_55 ;if it goes low at all, we reset the loop

DJNZ R2,PB\_65 ;this inner loop takes 7.8125mS

DJNZ R3,PB\_60 ;this outer loop takes 21.54 mS

;

PB\_90:

SETB EA ;resume interrupts

CLR IE0 ;clear the interrupt flag

POP PSW ;this restores register bank 0

RETI

;

;---------------------------------------------------------------------------

;

; LONG INTERVAL TIMER INTERRUPT -- this interrupt is the main pacing

; element of this application. This interrupt occurs every 0.5 seconds

; while in the GUI and Display mode. In the Logging mode, the interval

; is extended to 10 minutes.

;

; Within the interrupt routine, we'll handle the conversion process, but

; defer the processing of data (either displaying, transmitting, or logging)

; to the main program loop, which will be informed of the interrupt via

; a semaphore. The reason for this is to provide a place where we can

; manipulate the power control bits readily, since we'll need a place

; to return to at the conclusion of the interrupt.

;

;---------------------------------------------------------------------------

;

TICINT:

PUSH PSW ;and the machine

PUSH ACC ;save acc state

SETB RS0 ;we'll use the second register bank

;

; Clear the interrupt source flag so we're ready for the next pass

;

MOV A,TIMECON ;we need to clear the interrupt flag

ANL A,#0FBH ;clear the TII

MOV TIMECON,A ;put the byte back

;

; We need to kick off a single conversion here

;

MOV ADCMODE,#32H ;kick off a single conversion of the

;primary and auxilliary A/D

JNB RDY0,$ ;and wait for completion

SETB CNVREADY ;signal the main loop that the conversion

;is ready

; Are we in the logging mode?

;

JNB LOGMODE,TIC\_20 ;if we're in log mode, the interrupt will only

;occur every 10 minutes, so we don't need to

;check the minutes register

SETB LOGFLG

POP ACC ;restore the accum

POP PSW ;and the PSW (restores RB0)

RETI

;

;----------------------

;

; While we're in the display/GUI mode, we still want to log every five minutes

; to data FLASH. To do this, we'll check the long interval counter minutes

; register; when it reaches 5, we'll set a semaphore to signal the main program

; loop to process a log point, and clear the minutes register

;

; OK, we're in the display mode, so we'll need to check the MIN register

;

TIC\_20:

MOV A,MIN ;read the elapsed time register

CJNE A,#5,TIC\_70 ;not 5 minutes yet?

;

; OK, we've reached 5 minutes, time to reset the MIN register

;

MOV A,TIMECON ;we'll need to disable the TCEN

ANL A,#0FEH ;so we'll clear the bit

MOV TIMECON,A

MOV MIN,#0

SETB ACC.0 ;turn the bit back on

MOV TIMECON,A ;and re-write TIMECON

SETB LOGFLG ;set bit to indicate it's time to log data

;

; If we have been sitting in the display/GUI mode for 10 minutes, it's time to

; switch to the logging mode. We'll keep track of how often we log data while in

; the display mode, and if we've been doing it for 10 minutes, AND no serial

; port interrupt or switch interrupt has occurred, we'll drop into the logging

; mode

;

INC LOGCTR ;increment to see how long we've been

MOV A,LOGCTR ;in the display/gui mode

CJNE A,#2,TIC\_70 ;if not 10 minutes, we'll just return

;

;----------------------

;

; OK, we've been doing this for ten minutes... time to go to the logging

; mode. We'll set the bit, and also reprogram the TIC for 5 minute interrupt

; intervals, and we'll show 'Log' on the display

;

SETB LOGMODE ;set the log mode flag

MOV TIMECON,#63H ;set for one minute timebase

MOV INTVAL,#5 ;interrupt every 5 minutes

;

; While we're in the logging mode, we'll blank the display

;

MOV LCDMSB,#0H

MOV LCD2SB,#0H

MOV LCD3SB,#0H

MOV LCDLSB,#0H

LCALL DSPLSHIFT

;

TIC\_70:

POP ACC ;restore the accum

POP PSW ;and the PSW (restores RB0)

RETI

;

;---------------------------------------------------------------------------

;

; SERIAL PORT INTERRUPT -- this interrupt occurs whenever the serial port

; receives a character

;

;---------------------------------------------------------------------------

;

SPINT:

;

; The data output routines are designed to use polling, rather than an interrupt

; driven structure. Since the 8051 architecture doesn't provide for independent

; xmit and rcv interrupts, and since we need the rcv interrupt for incoming

; commands, we'll use a semaphore to convey the TI info to output routines

;

JNB TI,SPINT\_10 ;if it wasn't a transmit interrupt, we'll continue

SETB TXRDY ;if it was, we'll set it's alias

CLR TI ;and clear the actual bit

SJMP SPINT\_20 ;and return from the interrupt

;

SPINT\_10:

;

; In this application, the only incoming communications consists of

; single character commands. We're expecting one, and only one character

; to arrive. It will be more convenient to service the commands within

; the main program loop, so we'll just queue the command into CMDCHAR, set

; the CMDFLG, and return

;

MOV CMDCHAR,SBUF ;read the character from the UART

SETB CMDFLG ;set the semaphore to signal that a command awaits

CLR RI ;clear the flag in preparation for the next command

MOV LOGCTR,#0 ;any rcv int resets the timer for switching to log

;

SPINT\_20:

RETI

;

;---------------------------------------------------------------------------

;

; CMD\_SERVICE -- routine which services incoming commands received over

; the serial port. The incoming command charactor is matched (using a table)

; against the list of recognized commands, in both upper and lower case

; variants. Then, control is passed to the associated execution routine.

;

; (NOTE: with so few commands, we could just as easily done a brute force

; in-line comparison, rather than in indexed table structure, but this is

; better programming practice, and makes it easier to add commands. The

; total # of bytes would have been similar)

;

;---------------------------------------------------------------------------

;

CMD\_SERVICE:

;

CLR CMDFLG ;clear the command flag

MOV R2,#8 ;max number of commands

MOV DPTR,#CMDTAB ;point to the command table

CS\_10:

CLR A ;ready to fetch the command prototype

MOVC A,@A+DPTR ;fetch the prototype

CJNE A,CMDCHAR,CS\_20 ;jump if this isn't it

;

; We've found it, so let's grab the address of the command and

; vector to it

;

INC DPTR ;point to the high byte

CLR A ;ready to vector there

JMP @A+DPTR ;and go do it!

CS\_20:

INC DPTR ;point to the next command

INC DPTR

INC DPTR

DJNZ R2,CS\_10 ;and try the next one

;

; If it's not in the list of commands, we'll abort

;

RET

;

;-----------

;

; CMDTAB -- a table of command characters, and their associated

; execution vectors

;

CMDTAB:

DB 'C' ;switch to celsius mode

SJMP DO\_CELSIUS

DB 'c' ;ditto

SJMP DO\_CELSIUS

DB 'F' ;switch to fahrenheit mode

SJMP DO\_FAHRENHEIT

DB 'f' ;ditto

SJMP DO\_FAHRENHEIT

DB 'T' ;toggle between chip and RTD temp

SJMP DO\_SENSORTOGGLE

DB 't' ;ditto

SJMP DO\_SENSORTOGGLE

DB 'L' ;download log data

SJMP DO\_DOWNLOAD

DB 'l' ;ditto

SJMP DO\_DOWNLOAD

;

;-----------

;

; Command execution routines

;

DO\_CELSIUS:

CLR CFFLG ;set for celsius operation

RET

;

DO\_FAHRENHEIT:

SETB CFFLG ;set for fahrenheit operation

RET

;

DO\_SENSORTOGGLE:

;

; If no RTD exists, we won't toggle the SNSRFLG

;

JB RTDEXISTS,DO\_SEN\_10 ;allow toggle if RTD exists

RET ;else, just return

;

DO\_SEN\_10:

CPL SNSRFLG ;switch to the opposite sensor

RET

;

;---------------------------------------------------------------------------

;

;

; The DO\_DOWNLOAD routine transmits all logged data out the serial

; port. First, the data points themselves are transmitted, using a

; format which includes the data point number. The format is as

; follows:

;

; L<src>:<data><C or F><cr><lf>

;

; For example,

;

; 'LR:+023.4C'

;

; indicates a temperature of +23.4 degrees C, logged from the RTD sensor.

; The entire buffer (256 points) will be emitted from the board, in forward

; order (oldest data first) followed by one more record which tells

; the GUI application that log downloading is complete:

;

; 'XXX'<cr><lf>

;

DO\_DOWNLOAD:

;

; We'll disable the TIC interrupt while we're sending the download

;

; SETB DWNLDFLG ;we'll use a flag to hold off any TIC ints (debug)

; MOV IEIP2,#00H ;shut down the TIC interrupt

CLR EA ;turn off ALL interrupts

;

MOV SBUF,#0DH ;xmit a CR to get the UART in the correct state

; We'll also save the state of SNSRFLG

;

MOV C,SNSRFLG ;save the sensor flag

MOV TEMPSNSRFLG,C

;

; first, we need to fetch the control parameters;

;

MOV EADRL,#LOGCONTROL ;point to the control parameters

MOV ECON,#01H ;read them

;

; The LOGPTR is a pointer to the next available byte pair for logging.

; Since it's a circular 256 point (512 byte) buffer, the start of the

; buffer is at the very next location. We'll xmit the data points

; forwards, starting at the next available buffer location.

;

; LOGPTR points to the next sample, based on a 256 sample buffer. Since

; there are two samples per page, the page address consists of the upper

; 6 bits of LOGPTR. LOGPTR.1 points to the specific sample within the

; page. DNLDNDX serves to buffer LOGPTR, for the purposes of this loop.

;

MOV DNLDCTR,#0 ;256 sample count

MOV A,LOGPTR ;fetch the LOGPTR

MOV DNLDNDX,A ;and save it in an index

;

DO\_DOWN\_10:

RR A ;rotate one bit to define the page

ANL A,#07FH

MOV EADRL,A

MOV ECON,#01H ;read the page

;

; the LSB of DNLDNDX tells us whether the datapoint we want is in the

; first or second byte pair

;

JB DNLDNDX.0,DO\_DOWN\_20 ;skip if it's the second pair

MOV LOGDATHI,EDATA1 ;fetch the first pair

MOV LOGDATLO,EDATA2

SJMP DO\_DOWN\_30

DO\_DOWN\_20:

MOV LOGDATHI,EDATA3 ;fetch the second pair

MOV LOGDATLO,EDATA4

;

;

; Now we have the data point, and can xmit it

;

DO\_DOWN\_30:

;

LCALL LOGDATOUT

;

; We'll adjust the indices and loop until all 256 have been xmitted

;

INC DNLDNDX

MOV A,DNLDNDX ;point to the index

DJNZ DNLDCTR,DO\_DOWN\_10 ;repeat 256 times

;

;

; Once all 256 have been xmitted, we'll transmit the terminating record

;

;

LCALL CHAROUT

MOV A,#'X'

LCALL CHAROUT

MOV A,#'X'

LCALL CHAROUT

MOV A,#'X'

LCALL CHAROUT

MOV A,#0DH

LCALL CHAROUT

MOV A,#0AH

LCALL CHAROUT

;

; We'll restore the sensor flag

;

MOV C,TEMPSNSRFLG ;restore the sensor flag

MOV SNSRFLG,C

;

; And also re-enable the TIC interrupt

;

; MOV IEIP2,#04H ;enable the TIC interrupt counter

; CLR DWNLDFLG ;ok to allow TIC's again

SETB EA ;OK to turn on interrupts now

;

RET

;

;---------------------------------------------------------------------------

;

; LOGDATOUT -- a routine which spits out the logged data from data FLASH

; to the GUI via the serial port, in resonse to the reception of a 'L' or

; 'l' character on the serial port. This is called from the 'DO\_DOWNLOAD'

; routine.

;

;---------------------------------------------------------------------------

;

LOGDATOUT:

;

; The first order of business is to strip out the snsrflg bit, and

; sign-extend back into it's place

;

MOV C,LOGDATHI.7 ;fetch the source bit

MOV SNSRFLG,C ;and reserve it

;

; Now sign extend to fix the data

;

MOV C,LOGDATHI.6

MOV LOGDATHI.7,C

;

;-----------

;

; Now we can convert to packed BCD in preparation for creating the

; output strings

;

; First, we'll see if the argument is negative. If it is, we'll reserve

; the sign in SGNBIT and take the two's complement

;

;

LGO\_05:

MOV C,LOGDATHI.7 ;fetch the sign of the argument

MOV SGNBIT,C ;and save it

JNC LGO\_10 ;and don't complement if it's positive

;

; it's negative, so we'll take the two's complement

;

MOV A,LOGDATLO ;fetch the low byte

CPL A ;complement it

ADD A,#1 ;add one

MOV LOGDATLO,A ;and replace it

MOV A,LOGDATHI ;now fetch the high byte

CPL A ;complement it

ADDC A,#0 ;add any carry

MOV LOGDATHI,A ;and replace it

;

; we can now convert to packed BCD

;

LGO\_10:

MOV R1,#LOGDATHI ;point to the argument

MOV R0,#PBCDBUF ;and point to the output buffer

CALL INT\_TO\_BCD ;convert to packed BCD

;

; Now we can construct the output string and xmit it

;

MOV A,#'L' ;xmit the opening character

LCALL CHAROUT

;

JB SNSRFLG,LGO\_20 ;jump if RTD

MOV A,#'C' ;xmit C for chip

LCALL CHAROUT

;

SJMP LGO\_25

LGO\_20:

MOV A,#'R' ;xmit C for chip

LCALL CHAROUT

;

; Now send the exclamation mark

;

LGO\_25:

MOV A,#'!'

LCALL CHAROUT

;

;

LGO\_35:

JB SGNBIT,LGO\_40 ;a 1 indicates negative

MOV A,#'+' ;send the plus sign

LCALL CHAROUT

SJMP LGO\_50

LGO\_40:

MOV A,#'-'

LCALL CHAROUT

;

; Now we can send the three most significant digits

;

LGO\_50:

MOV A,PBCDBUF+1 ;fetch the top two digits

PUSH ACC ;save it, we'll need it again

SWAP A ;position the MSD

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHAROUT

;

POP ACC ;get the digit pair back

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHAROUT

;

MOV A,PBCDBUF ;fetch the bottom two digits

PUSH ACC ;save it, we'll need it again

SWAP A ;position the MSD

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHAROUT

;

; We need to insert the decimal point here

;

MOV A,#'.'

LCALL CHAROUT

;

; now the LSD

;

POP ACC ;get the digit pair back

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHAROUT

;

; Next, we transmit 'C'

;

MOV A,#'C' ;indicate celsius

LCALL CHAROUT

;

; Finish up with the carriage return and line feed

;

LGO\_80:

MOV A,#0DH

LCALL CHAROUT

MOV A,#0AH

LCALL CHAROUT

;

; Delay a bit here

;

; MOV R2,#0

; DJNZ R2,$

; DJNZ R2,$

; SETB TXRDY

RET

;

;---------------------------------------------------------------------------

;

; CHAROUT -- transmits a single character out the serial port. This is

; a pure polling routine, used during downloading, when all interrupts

; are disabled

;

;---------------------------------------------------------------------------

;

CHAROUT:

JNB TI,$ ;wait for UART availability

CLR TI ;clear the flag for next time

MOV SBUF,A ;send the character

MOV R2,#0

DJNZ R2,$

; MOV R2,#0

; DJNZ R2,$

RET

;

;

;---------------------------------------------------------------------------

;

; CHOUT -- transmits a single character out the serial port. This is

; essentially a polling routine, utilizing the semaphore from the

; serial interrupt which indicates the availability of the UART, and is

; used for all serial functions EXCEPT the download

;

;---------------------------------------------------------------------------

;

CHOUT:

JNB TXRDY,$ ;wait for UART availability

CLR TXRDY ;clear the flag for next time

MOV SBUF,A ;send the character

RET

;

;---------------------------------------------------------------------------

;

; LIVEDATAOUT -- this routine transmits one datum of live temperature data

; using polled transmission. It is used during the GUI mode to send

; the current temperature data to the PC.

;

; The data format is intended to be readable via Hyperterminal or an

; equivalent 'dumb terminal' program, as well as being interpretable by

; the GUI application. Each temperature is transmitted as one line of ASCII

; characters, terminated by a carriage return and line feed. The beginning

; character is a 'C' or 'R', denoting whether the temperature is from the

; on-chip sensor, or the RTD sensor. This is followed by a colon, then the

; temperature, in '+XXX.X' format, then 'C' or 'F', then the carriage return

; or line feed. For example:

;

; C:+027.3C (cr)(lf)

; R:-003.2F (cr)(lf)

;

; The routine presumes that the data, in packed BCD format, still resides

; in the PBCDBUF; it is intended that the LIVEDATAOUT routine is called just

; after the DISPLAY\_FORMAT routine is called

;

;

LIVEDATAOUT:

;

; First, transmit the 'C' or 'R', based on the state of SNSRFLG

;

JB SNSRFLG,LDO\_20 ;is it RTD? Then jump

MOV A,#'C' ;it's the chip temperature

LCALL CHOUT

SJMP LDO\_30

LDO\_20:

MOV A,#'R'

LCALL CHOUT

;

; Now send the colon

;

LDO\_30:

MOV A,#':'

LCALL CHOUT

;

; Now send the sign character

;

LDO\_35:

JB SGNBIT,LDO\_40 ;a 1 indicates negative

MOV A,#'+' ;send the plus sign

LCALL CHOUT

SJMP LDO\_50

LDO\_40:

MOV A,#'-'

LCALL CHOUT

;

; Now we can send the three most significant digits

;

LDO\_50:

MOV A,PBCDBUF+1 ;fetch the top two digits

PUSH ACC ;save it, we'll need it again

SWAP A ;position the MSD

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHOUT

;

POP ACC ;get the digit pair back

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHOUT

;

MOV A,PBCDBUF ;fetch the bottom two digits

PUSH ACC ;save it, we'll need it again

SWAP A ;position the MSD

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHOUT

;

; We need to insert the decimal point here

;

MOV A,#'.'

LCALL CHOUT

;

; now the LSD

;

POP ACC ;get the digit pair back

ANL A,#0FH ;isolate the LS nibble

ORL A,#30H ;convert to ASCII

LCALL CHOUT

;

; Next, we transmit 'C' or 'F'

;

JB CFFLG,LDO\_70 ;is it fahrenheit? Then jump

MOV A,#'C' ;indicate celsius

LCALL CHOUT

SJMP LDO\_80

LDO\_70:

MOV A,#'F'

LCALL CHOUT

;

; Finish up with the carriage return and line feed

;

LDO\_80:

MOV A,#0DH

LCALL CHOUT

MOV A,#0AH

LCALL CHOUT

;

RET

;

;---------------------------------------------------------------------------

;

; TEMP\_CONVERT -- converts a temperature (stored as a twos complement

; integer in TEMP|TEMP+1) into packed BCD within the PBCDBUF

;

;---------------------------------------------------------------------------

;

; First, we'll see if the argument is negative. If it is, we'll reserve

; the sign in SGNBIT and take the two's complement

;

TEMP\_CONVERT:

;

MOV C,TEMP.7 ;fetch the sign of the argument

MOV SGNBIT,C ;and save it

JNC TC\_10 ;and don't complement if it's positive

;

; it's negative, so we'll take the two's complement

;

MOV A,TEMP+1 ;fetch the low byte

CPL A ;complement it

ADD A,#1 ;add one

MOV TEMP+1,A ;and replace it

MOV A,TEMP ;now fetch the high byte

CPL A ;complement it

ADDC A,#0 ;add any carry

MOV TEMP,A ;and replace it

;

; we can now convert to packed BCD

;

TC\_10:

MOV R1,#TEMP ;point to the argument

MOV R0,#PBCDBUF ;and point to the output buffer

;

; The INT\_TO\_BCD is destructive of it's input argument, so we'll

; temporarily save the argument on the stack

;

PUSH TEMP

PUSH TEMP+1

CALL INT\_TO\_BCD ;convert to packed BCD

POP TEMP+1

POP TEMP

;

RET

;

;---------------------------------------------------------------------------

;

; DISPLAY\_FORMAT -- routine which takes a 16 bit integer argument

; and writes it to the LCD display. The argument is presumed to be

; a signed two's complement argument in the range of -25C to 85C, or

; -13F to 185F. The argument is found at TEMP (2 bytes, high byte first)

;

;---------------------------------------------------------------------------

;

DISPLAY\_FORMAT:

;

;

; We're ready to load up the LCD display buffer locations, based

; on the contents of the PBCDBUF. First, let's deal with the F or C

; character in the LSD position. The MSB of this buffer byte must also be

; set to display the decimal point, and is included in the definitions of

; FCHAR and CCHAR

;

JB CFFLG,DF\_20 ;jump if fahrenheit

MOV LCDLSB,#CCHAR ;put the 'C' character there

SJMP DF\_30

DF\_20:

MOV LCDLSB,#FCHAR ;put the 'F' character there

;

;

; if we're overflowed or underflowed, we'll write the appropriate fixed

; display pattern (e.g., ' --.-F') to the display

;

JNB OVERFLOW,DF\_24 ;are we overflowed?

MOV DPTR,#OVERSEG ;yes, so write the overflow pattern

SJMP DF\_26

DF\_24:

JNB UNDERFLOW,DF\_30 ;are we underflowed?

MOV DPTR,#UNDERSEG ;yes, so write the underflow pattern

DF\_26:

MOV R0,#LCDMSB ;point to the LCD buffer

MOV R1,#3 ;three bytes to write

DF\_27:

CLR A

MOVC A,@A+DPTR ;fetch the pattern byte

MOV @R0,A ;write it

DJNZ R1,DF\_27 ;repeat for all three

SJMP DF\_40

;

; Now we'll deal with each digit, decoding the corresponding BCD nibble

; to it's seven segment code before loading it into the buffer

;

DF\_30:

MOV A,PBCDBUF ;fetch the lowest two digits

PUSH ACC ;save it, we'll need it again

ANL A,#0FH ;isolate the LS nibble

MOV DPTR,#SEGTAB ;point to the segment table

MOVC A,@A+DPTR ;and fetch the segment code

MOV C,SGNBIT ;the MSB if this buffer controls the

MOV ACC.7,C ;minus sign

MOV LCD3SB,A ;load the segment code to the LCD buffer

;

POP ACC ;get the packed BCD argument back

SWAP A ;isolate the other nibble

ANL A,#0FH ;isolate the LS nibble

MOV DPTR,#SEGTAB ;point to the segment table

MOVC A,@A+DPTR ;and fetch the segment code

MOV LCD2SB,A ;load the segment code to the LCD buffer

;

MOV A,PBCDBUF+1 ;fetch the next two digits

PUSH ACC ;save it, we'll need it again

ANL A,#0FH ;isolate the LS nibble

MOV DPTR,#SEGTAB ;point to the segment table

MOVC A,@A+DPTR ;and fetch the segment code

MOV LCDMSB,A ;load the segment code to the LCD buffer

;

POP ACC ;get the packed BCD argument back

SWAP A ;swap nibbles

ANL A,#0FH ;isolate the digit

JZ DF\_40 ;if zero, we won't need the '1'

MOV A,LCD2SB ;else we'll need to set the MSB of LCD2SB

ORL A,#80H ;to display the 'half digit'

MOV LCD2SB,A ;now re-load the buffer

;

DF\_40:

CALL DSPLSHIFT ;finally, we can write the display

RET

;

; Patterns for overflow and underflow, in LCDMSB/LCD2SB/LCD3SB order

OVERSEG: DB 040H,040H,040H

UNDERSEG: DB 040H,040H,0C0H

;

;--------------------------------------------------------------------------

;

; DSPLSHIFT -- Display shifting routine. This shifts the contents of the

; display segment buffers out to the MM5483 LCD display controller. The

; contents of LCDLSB through LCDMSB are shifted out, with clock, least

; significant bit first. Since the shift process is much faster than the

; LCD excitation rate, there's no need to use the latch feature of the

; MM5483; no flicker will result.

;

;--------------------------------------------------------------------------

;

DSPLSHIFT:

MOV R1,#4 ;four bytes to shift

MOV R0,#LCDLSB ;point to the LSB of the display data

DSPL\_10:

MOV A,@R0 ;fetch the display data

;

; We need to transpose the bits because the SPI shifts MSB first, and the

; display controller chip is wired for LSB first

;

MOV R2,#8 ;eight bits to transpose

DSPL\_50:

RRC A ;rotate to expose bit in the carry

XCH A,B ;we'll rotate it into the B reg

RLC A

XCH A,B

DJNZ R2,DSPL\_50

MOV A,B ;the transposed data was in B

;

; OK, we're ready to shift

;

MOV SPIDAT,A ;load the data to the SPI

JNB ISPI,$ ;wait for the transfer to complete

CLR ISPI ;ok, clear it for the next byte

DEC R0 ;point to next byte

DJNZ R1,DSPL\_10 ;do all fout

RET

;

;---------------------------------------------------------------------------

;

; MAD\_AB\_DPTR -- multiply A times B and add to DPTR. This encapsulation

; of a repetitively coded sequence is a big convenience for table lookup

; mechanizations

;

;---------------------------------------------------------------------------

;

MAD\_AB\_DPTR:

MUL AB ;multiply a times b

ADD A,DPL ;add the low byte

MOV DPL,A ;and save the low byte

MOV A,B ;get the high byte

ADDC A,DPH ;add the high byte

MOV DPH,A ;and return it

RET

;

;---------------------------------------------------------------------------

;

; INT\_TO\_BCD -- converts an integer to packed BCD format. The basic

; approach is a left shift algorithm. Each time the integer is shifted,

; we add whatever bit is shifted, and multiply the BCD accumulator by two,

; decimal-adjusting as we go along. R0 points to the target (output),

; which is three bytes, R1 points to the integer (two bytes). The order of

; bytes in the integer argument is Hi|Lo.

;

; In the context of this program, we'll always be converting positive

; integers (the sign bit will be taken care of separately).

;

;---------------------------------------------------------------------------

;

INT\_TO\_BCD:

;

MOV R2,#16 ;16 bits bits to convert

;

; Clear the target

;

MOV @R0,#0 ;fill with zeroes

INC R0

MOV @R0,#0

INC R0

MOV @R0,#0

DEC R0

DEC R0

;

; OK, we've now got a positive integer in the argumnent. We'll need to

; rotate the source integer, generating a potential carry (R1 points to

; the integer

;

ITB\_10:

INC R1 ;point to the lowest byte

MOV A,@R1 ;get the low byte

RLC A ;rotate left

MOV @R1,A ;replace it

DEC R1 ;do the middle byte

MOV A,@R1 ;get the middle byte

RLC A ;rotate it with the carry from the low byte

MOV @R1,A ;and replace it

;

; Now double the target, add the carry, and perform the decimal adjusts

;

MOV A,@R0 ;get the byte

MOV B,A ;copy it to B

ADDC A,B ;double it and add the carry

DA A ;decimal adjust it

MOV @R0,A ;replace the target byte

INC R0 ;next byte

;

MOV A,@R0 ;get the byte

MOV B,A ;copy it to B

ADDC A,B ;double it and add the carry

DA A ;decimal adjust it

MOV @R0,A ;replace the target byte

INC R0 ;next byte

;

MOV A,@R0 ;get the byte

MOV B,A ;copy it to B

ADDC A,B ;double it and add the carry

DA A ;decimal adjust it

MOV @R0,A ;replace the target byte

;

DEC R0 ;restore the pointer

DEC R0

;

DJNZ R2,ITB\_10 ;do this 16 times

;

RET

;

;---------------------------------------------------------------------------

;

; SEGTAB -- a table of byte values which activate the appropriate

; segments of the digits in the seven segment LCD display

;

SEGTAB:

DB 03FH ;zero

DB 003H ;one

DB 05EH ;two

DB 04FH ;three

DB 063H ;four

DB 06DH ;five

DB 07DH ;six

DB 007H ;seven

DB 07FH ;eight

DB 06FH ;nine

;

;---------------------------------------------------------------------------

;

; CHIPTEMPSCALE -- routine which scales the on-chip temperature sensor

; result.

;

; At the registers of the auxilliary A/D, the scaling of the on-chip sensor

; is in units of 1/256 of a degree C (one LSB of the high byte = 1 degree C).

; In order to resolve 0.1 degree, in either C or F, we need 14 raw bits, in

; which case, an LSB will be equal to 0.03125 degrees C, or 0.5625 degrees F.

;

CHIPTEMPSCALE:

;

; The first order of business is to fetch the conversion result, convert

; to two's complement, and shift to yield a 14 bit number

;

MOV R2,#2 ;two bit shift

CTS\_10:

MOV A,TEMP ;fetch the high byte

RRC A ;rotate right

MOV TEMP,A ;and save

MOV A,TEMP+1 ;now the low byte

RRC A ;rotate right

MOV TEMP+1,A ;and save

DJNZ R2,CTS\_10 ;do it twice

;

; Now sign extend

;

MOV C,TEMP.5

MOV TEMP.7,C

MOV TEMP.6,C

;

; Our number is now a signed 14 bit number, with a resolution of 1/64

; degree C. For conversion to tenths of a degree C scaling, all we need to

; do is multiply. For conversion to tenths of a degree fahrenheit, we'll

; need to both multiply and add.

;

; For degrees C, we can multiply the argument by 10246, which will result

; in a 4 byte product equal to the temperature, in tenths of a degree,

; scaled by 2^16 (65536). Our desired result will therefore be found in the

; upper two bytes of the product.

;

MOV R3,TEMP ;position the data, high byte first

MOV R2,TEMP+1

MOV DPTR,#CCHIPSCALE ;point to the constant

CLR A

MOVC A,@A+DPTR ;fetch it

MOV R1,A ;and deposit it in the argument

MOV A,#1

MOVC A,@A+DPTR ;same for the low byte

MOV R0,A

LCALL MUL16

;

; The result of the operation will be the highest two bytes of the product,

; located in R3|R2

;

MOV TEMP,R3 ;fetch back the scaled data

MOV TEMP+1,R2

;

; If we're in the Fahrenheit mode, we can continue on to convert to Fahrenheit

;

JB TEMPCFFLG,CTS\_20 ;continue if fahrenheit

RET ;else we're done

;

; Conversion of celsius to fahrenheit is (C\*1.8)+32... however, we're scaled

; in units of tenths of a degree C. Because of the magnitudes, we can't use the

; same trick of re-scaling by 2^16. The most convenient way to rescale is to

; multiply by 7373, which produces a result, scaled in tenths of a degree,

; multiplied by 4096 (2^12)... we merely need to shift right by 12 bits,

; add the offset term (320), and we'll have our number.

;

; The data is already in R3|R2, so we'll just load R1|R0 with the scaling

; constant, and multiply

;

CTS\_20:

MOV R1,#HIGH 7373 ;load the scaling value

MOV R0,#LOW 7373

LCALL MUL16

;

; Now we'll 'shift' by 12 bits...

;

MOV R7,#12 ;twelve bit shift

;

CTS\_25:

MOV A,R3 ;top byte

CLR C

RRC A

MOV R3,A

MOV A,R2 ;second byte

RRC A

MOV R2,A

MOV A,R1 ;third byte

RRC A

MOV R1,A

MOV A,R0 ;least significant byte

RRC A

MOV R0,A

DJNZ R7,CTS\_25

;

MOV TEMP,R1 ;transfer back to the TEMP reg pair

MOV TEMP+1,R0

;

; Almost done... we just need to add #320

;

MOV A,TEMP+1

ADD A,#LOW 320 ;add the low byte

MOV TEMP+1,A ;save it

MOV A,TEMP ;now the hugh byte

ADDC A,#HIGH 320 ;add any carry

MOV TEMP,A ;and replace it

;

RET

;

CCHIPSCALE: DW 10246 ;scale factor for chip temp scaling in C

;---------------------------------------------------------------------------

;

;

; A NOTE ON LINEARIZATION: While there are many different approaches

; to linearization, each approach has tradeoffs in terms of size,

; complexity, coding difficulty, and execution time. The simplest

; forms of linearization involve 'brute force' table lookups; the

; most complex (but most accurate) usually employ some form of

; polynomial approximations, achieved with floating point math.

;

; For this project, we have chosen a segmented linearization approach,

; which approximates the transfer curve of the sensor via a number of

; straight line segments which closely follow the shape of the curve.

; Segment-based linearizers can be reasonably fast and small. This one

; is especially small; the code requires only 87 bytes, and each

; table (one for celsius, one for fahrenheit) require only 51 bytes

; each.

;

; The required temperature range is divided up into a number of segments

; of equal A/D code length (which makes the calculation convenient,

; especially when the segments are aligned on natural binary borders, as

; is the case here). For each segment, the table provides an 'm'

; term and a 'b' term, in order to compute the familiar 'f=mx+b'

; equation. The coefficients are scaled so that the result of the

; linearization process is an integer in units of tenths of a degree,

; ready for display or transmission.

;

; The segment size was carefully considered, so that the resulting

; conformance error (deviation between the output and the actual temp)

; would be smaller than the desired output resolution.

;

; Each segment record consists of two numbers. The first number is the

; temperature at the base of the segment, in tenths of a degree. For

; example, the base temperature of the third segment of the 'degrees C'

; table is -147, or -14.7 degrees C. The second number reflects the number

; of tenths of a degree between the base of this segment, and the base of

; the next segment. In this example, it is 6.3 degrees.

;

; By aligning the segments on natural binary borders (in this case, 128

; byte segments, when treating the A/D result as a 13 bit number),

; The segment of interest is determined simply by taking the 13 bit A/D

; result, masking out the lower 7 bits, subtracting 4608 (the code of

; the start of the first segment), and shifting the result to the right

; by 7 bits to form the segment index.

;

; Once we've found the segment, we subtract the segment base temperature

; from the A/D output to find the difference value, which is then multiplied

; with the segment gain (the byte value which follows the segment base

; temperature), and shift the result to the right by 7 bits to normalize it.

; This value is then added to the segment base temperature to produce

; the linearized temperature.

;

; For additional convenience, we'll use a separate but similar table

; for both degrees C and degrees F, thereby eliminating any multibyte

; multiplication whatsoever.

;

; (ADDITIONAL NOTE: the computations which produced these tables depend on

; the assumptions about A/D scaling, detailed elsewhere in this code. If

; there are any changes to those assumptions, then these tables would have

; to be re-calculated)

;

;---------------------------------------------------------------------------

;

; ADCLIN -- linearizes the output of the A/D converter, for RTD sensors.

; The routine presumes that DPTR points to the linearization table of

; interest, and TEMP|TEMP+1 (in hi-lo order) holds the conversion result

;

;---------------------------------------------------------------------------

;

ADCLIN:

;

; First, we'll compute the table index. Only the top 13 bits are of

; interest, and of those, the top 7 represent the table index

;

CLR UNDERFLOW ;clear the error flags

CLR OVERFLOW

;

; First, we need to shift the ADC data to truncate it to 13 bits. The

; data is 15 bits, so we need to shift by two

;

MOV R2,#2 ;two bit shift

ADCLIN\_05:

CLR C

MOV A,TEMP ;get the high byte

RRC A ;shift it

MOV TEMP,A

MOV A,TEMP+1 ;now the mid byte

RRC A ;shift it

MOV TEMP+1,A

DJNZ R2,ADCLIN\_05

;

; The top 6 bits, offset by 24H, represent the table index. These bits

; constitute the index of 128 byte segments in the table

;

MOV C,TEMP+1.7 ;ready to shift up the MSB of the 2nd byte

MOV A,TEMP ;get the upper byte

RLC A ;shift it

CLR C ;ready to offset the index

SUBB A,#24H ;subtract to normalize the index

MOV R3,A ;save this index

;

; if this resulted in a carry, we've underflowed

;

JNC ADCLIN\_10 ;it's an underflow

SETB UNDERFLOW ;so tag it

RET ;and return

;

ADCLIN\_10:

MOV A,R3 ;see if it's an overflow

CLR C

SUBB A,#17 ;if it's greater than 17 (max # of segments),

JC ADCLIN\_20 ;it's an overflow,

SETB OVERFLOW ;so tag it

RET ;and return

;

; The residue is the bottom 7 bits.

;

ADCLIN\_20:

MOV A,TEMP+1 ;get the lower byte

ANL A,#07FH ;mask for the lowest 7 bits

PUSH ACC ;save this for a moment

;

; We'll need to fetch the magnitude (span) of the segment

; (DPTR is already pointing to the start of the table) and

; multiply it by the residue

;

MOV A,R3 ;get the table index back

MOV B,#3 ;multiply by three (three bytes per table entry)

LCALL MAD\_AB\_DPTR ;find the product and add to the dptr

MOV A,#2 ;it's the third byte we want

MOVC A,@A+DPTR ;fetch it!

MOV B,A ;save it in B

POP ACC ;get the residue back

MUL AB ;multiply the two

;

; We need to shift this product back to the right by 7 bits to normalize it

; to the scale.

;

MOV C,ACC.7 ;grab the top bit of the LSB

XCH A,B ;now get the upper byte

RLC A ;and rotate

;

; The ACC now has the scaled magnitude of the segment, which needs to be

; added to the segment base temp to give us the final temp

;

PUSH ACC ;save it for a moment

CLR A

MOVC A,@A+DPTR ;fetch the high byte of the base temp

MOV R6,A ;save in R6

MOV A,#1

MOVC A,@A+DPTR ;and fetch the low byte

MOV R7,A

;

; Now do the addition. We'll drop the result into TEMP|TEMP+1 for

; convenience

;

POP ACC ;get the magnitude term back

ADD A,R7 ;add to the low byte

MOV TEMP+1,A ;save it

MOV A,R6 ;now the upper byte, with carry

ADDC A, #0

MOV TEMP,A

RET

;

;--------------------------------------------------------------------

;

; 16 x 16 bit signed multiply routine follows (used by CHIPTEMPSCALE)

;

;--------------------------------------------------------------------

;

; MUL16 -- multiplies two signed integers, producing a 32 bit

; result. The approach taken converts the multiplicand and

; multiplier to sign-mag form (preserving the signs), performs

; a straight binary multiply, and then converts the product back

; to two's comp form, taking into account the signs of the

; operands. One operand is R1|R0, the other is R3|R2

;

;--------------------------------------------------------------------

;

MUL16:

LCALL TWOSCMP16\_TO\_MAG ; 2's comp -> Mag/Sign

LCALL TWOSCMP16\_TO\_MAG2 ; 2's comp -> Mag/Sign

LCALL MUL16X16

LCALL MAG32\_TO\_TWOSCMP ; Mag/Sign -> 2's Comp

RET

;--------------------------------------------------------------------

;

; TWOSCMP16\_TO\_MAG -- converts a 16 bit two's complement argument

; (in R3|R2) to magnitude/sign format. There's a copy which operates

; on R1|R0 just afterwards

;

;--------------------------------------------------------------------

;

TWOSCMP16\_TO\_MAG:

MOV A, R3 ;read high into accumulator

JB ACC.7, TTM\_16\_10 ;negative if bit 7 is 1

CLR SGNBITY ;clear sign bit if 'positive'

RET ;done

TTM\_16\_10:

SETB SGNBITY ;set sign flag

MOV A, R2 ;number is negative

CPL A ;complement

ADD A, #1 ;and add +1

MOV R2, A

MOV A, R3 ;get next byte

CPL A ;complement

ADDC A, #0

MOV R3, A

RET

;

; TWOSCMP16\_TO\_MAG2 is the same as just above, but operates on

; the other register pair (for convenience)

;

TWOSCMP16\_TO\_MAG2:

MOV A, R1 ;read high into accumulator

JB ACC.7, TTM2\_16\_10 ;negative if bit 7 is 1

CLR SGNBITX ;clear sign bit if 'positive'

RET ;done

TTM2\_16\_10:

SETB SGNBITX ;set sign flag

MOV A, R0 ;number is negative

CPL A ;complement

ADD A, #1 ;and add +1

MOV R2, A

MOV A, R1 ;get next byte

CPL A ;complement

ADDC A, #0

MOV R3, A

RET

;--------------------------------------------------------------------

;

; MAG32\_TO\_TWOSCMP -- converts a 32 bit sign-mag argument to

; two's comp form. The argument input/output is R3|R2|R1|R0.

; Since this is called as a result of a 16 bit operation, the

; individual signs of the arguments are tested to determine the

; sign of the result

;

;--------------------------------------------------------------------

;

MAG32\_to\_TWOSCMP:

JB SGNBITX, MTT\_10 ; test X sign

JB SGNBITY, MTT\_20 ; test Y sign

RET

MTT\_10:

JNB SGNBITY, MTT\_20

RET

MTT\_20:

MOV A, R0 ; negate number

CPL A ; complement

ADD A, #1 ; and add +1

MOV R0, A

MOV A, R1 ; get next byte

CPL A ; complement

ADDC A, #0

MOV R1, A

MOV A, R2 ; get next byte

CPL A ; complement

ADDC A, #0

MOV R2, A

MOV A, R3 ; get next byte

CPL A ; complement

ADDC A, #0

MOV R3, A

RET ; done

;

;

;--------------------------------------------------------------------

;

; MUL16X16 -- 16 by 16 bit unsigned multiply. The multiplicand is in

; R1|R0, and the multiplier is in R3|R2, the product is returned

; in R3|R2|R1|R0

;

;--------------------------------------------------------------------

;

MUL16X16:

;

MOV A, R0

MOV B, R2

MUL AB ; multiply XL x YL

PUSH ACC ; stack result low byte

PUSH B ; stack result high byte

;

MOV A, R0

MOV B, R3

MUL AB ; multiply XL x YH

;

POP 00H

ADD A, R0

MOV R0, A

CLR A

ADDC A, B

MOV DPL, A

MOV A, R2

MOV B, R1

MUL AB ; multiply XH x YL

ADD A, R0

MOV R0, A

MOV A, DPL

ADDC A, B

MOV DPL, A

CLR A

ADDC A, #0

;

PUSH ACC ; save intermediate carry

;

MOV A, R3

MOV B, R1

MUL AB ; multiply XH x YH

ADD A, DPL

MOV R2, A

POP ACC ; retrieve carry

ADDC A, B

MOV R3, A

MOV R1, 00H

POP 00H ; retrieve result low byte

RET

;

;

;---------------------------------------------------------------------------

;

; Linearization tables -- the following two tables support RTD

; linearization (see ADCLIN)

;

;---------------------------------------------------------------------------

;

; celsius linearization table

;

CLINTAB:

DW -273

DB 63

DW -210

DB 63

DW -147

DB 63

DW -84

DB 64

DW -20

DB 63

DW 43

DB 64

DW 107

DB 63

DW 170

DB 64

DW 234

DB 64

DW 298

DB 64

DW 362

DB 64

DW 426

DB 65

DW 491

DB 64

DW 555

DB 65

DW 620

DB 65

DW 685

DB 64

DW 749

DB 65

DW 814

DB 65

;

; fahrenheit linearization table

;

FLINTAB:

DW -172

DB 114

DW -58

DB 113

DW 55

DB 114

DW 169

DB 114

DW 283

DB 114

DW 397

DB 115

DW 512

DB 115

DW 627

DB 115

DW 742

DB 115

DW 857

DB 115

DW 972

DB 116

DW 1088

DB 115

DW 1203

DB 116

DW 1319

DB 117

DW 1436

DB 116

DW 1552

DB 117

DW 1669

DB 117

DW 1786

DB 117

;

;-------------------------------------------------------------

;

;TAB\_BASE EQU 4608

;TAB\_INCR EQU 128

;

;

END