IDI ASSIGNMENT

A Digital Clock

REPORT

Wade Tregaskis
02557793
5th of May, 2005

CONTENTS

1.0 Introduction

The assignment as given was to write assembly software for a digital clock, to operate on the PIC development boards provided by La Trobe University. This clock is required to keep reasonably accurate time, with one second resolution. It is required to display hours, minutes and seconds, in the form HH:MM and MM:SS, selectable by the user. It also must display a text message, which should be longer than can be displayed all at once, and should scroll at a suitable speed to be read. Lastly, it should have functionality to modify the time.

In addition to these requirements, the author added minor additional functionality, such as the option to display time in 12-hour or 24-hour format, as well as full HH:MM:SS. The scrolling message was also integrated with the time display, to provide a more pleasing appearance and functionality.

The sample code and tutorials provided by the PICTutor software provide a good basis for developing this program. In particular, the sample code for tutorials 29, 30 and 31 was analysed at the very beginning, as an example of how to approach the problem. No code from those tutorials, or any other, was directly copied into this assignment, save for common declarations (e.g. OPTIONS, W, F, etc).

2.0 Design

The system was built using a bottom-up methodology, developing first the basic display functionality, followed by use of the timer, etc. It was reviewed periodically and refactored as necessary. As an inexperienced PIC programmer, the author felt it was counter-productive to attempt to design the whole system at once, from scratch, when there were still unknowns as to how to develop for the PIC and development board.

2.1 Startup/Reset

When the program first executes, it jumps immediately to the SETUP procedure, which initialises the board as necessary - configuring I/O lines, initialising the LCD display, etc - and sets initial values for things such as the time, format & mode of display, etc.

Configuring the LCD display in particular takes some time - 40 timer cycles (approximately 500ms) are allowed from power-up to initialisation, which is necessary to allow the LCD display to internally initialise itself. After basic configuration another 40 timer cycles are allowed for the configuration to take proper effect - it takes significant time to clear all the LCDs internal memory, for example.

Once initialisation is complete, the message and the initial time value are displayed (more details in section 2.2), and the program enters the MAIN loop - discussed in section 2.3.

2.2 LCD Display

Programming the display was the hardest aspect of the assignment. The documentation provided, "How to Use Intelligent L.C.D.s" by Julyan Ilett, was helpful but not always specific. For example, it instructs the reader to issue multiple initialisation commands in sequence, before trying to issue any real commands, without any actual explanation as to why this is necessary, or how many dummy commands must be sent before the display is ready. Consequently, the laboratory code was most useful in reverse engineering the LCD display requirements. The code written by the author for LCD initialisation is thus effectively identical to the that provided in the laboratories, which is ultimately just a likely coincidence - much of the LCDs initialisation is fairly fixed, and the steps required for any given use are largely the same. The author did attempt to experiment with the LCD initialisation, but found little room for modification - while it should in theory be possible to initialise the LCD in just 4 commands, halving the size of the init table, the dummy initialisation commands are required in order for the LCD to respond to any subsequent commands.

Once the LCD is initialised (along with the rest of the board, as detailed in section 2.1) DISPLAY_MESSAGE is called. This function programs the first line of the LCD with the message stored in the LCD_MESSAGE lookup table. It is only ever called once, as to scroll the message the display window is shifted, rather than rewriting the message at a different offset. The reason for this is simple - the message on line one is to scroll, while the time on line two is to appear fixed. There are two ways to do this:

1. Redisplay both the message and the time as necessary, manually shifting the message to simulate scrolling, and keeping the time at a fixed location in memory.
2. Shift the display window on the LCD to provide scrolling over the message, and re-display the time for each shift at a new location, making the time appear fixed within the display window.

On the surface the former may seem simpler, because it is done entirely in the PIC with no fiddling of the LCDs display window and other settings. However, it requires the redisplay of the entire message each time - which may entail performing wrapping as well - which is rather wasteful for a static message. Given that the time display must change each second anyway - in MM:SS mode, at least - it is relatively inexpensive to perform the required adjustments at that time. In this way, the scrolling message is effectively performed automatically, and the main body of the code need only concern itself with the time.

To keep the time fixed within the display, the variable LCD_POSITION is used to keep track of the display window. It indicates the offset at which the time should start being written. Thus, each time the display is shifted to the right, LCD_POSITION is incremented to shift the time display to the right, to keep up. The DISPLAY_TIME function uses LCD_POSITION as it's start location when displaying the time - as discussed in section 2.4.

For using the LCD, several functions are provided - notably:

- LCD_OUTPUT_BYTE
  Outputs a single byte, provided to the function in W, to the LCD display. It does not explicitly modify the cursor position or other display attributes prior to doing so. It is also assumed the LCD_RS bit of the LCD_CONTROL file is set appropriately, for data or control operation.
- LCD_OUTPUT_CONTROL
  Outputs a single control byte, provided to the function in W, to the LCD display. This function simply manipulates the LCD_RS bit in LCD_CONTROL as appropriate, calling LCD_OUTPUT_BYTE to perform the actual output.
- LCD_OUTPUT_BCD
  Outputs two digits based on the byte passed to the function in W, which is assumed to contain two BCD digits in the high and low nibbles. This function primarily just selects each nibble appropriately, converts from "binary" to ASCII format for display, and calls LCD_OUTPUT_BYTE.
- DISPLAY_ERASE
  Simply erases N characters, where N is specified by the value passed to the function in W. It does this by outputting spaces at the current cursor position, performed by a loop which calls LCD_OUTPUT_BYTE.

2.3 The MAIN loop

The MAIN loop repeats itself infinitely. It's basic functionality is to check for user inputs (in the form of button presses), monitor the timer for overflow and keep track of the number of overflows recorded, and to increment the time when sufficient timer overflows have occurred.

MAIN uses the GET_INPUT function to detect any new key presses since it last looped through. It returns a bit map of the 5 port A buttons, with a newly pressed button corresponding to a 1 in that bit map. MAIN tests each of the five bits, and triggers the appropriate action if it is set - e.g. manually incrementing one of the fields, or changing the display mode, etc.

These actions are performed independently of other functionality, which is to monitor the timer and detect the passage of each second of time. Each time the timer overflows, the variable CLOCK_COUNT is decremented. If CLOCK_COUNT is still not zero, MAIN loops again. When it does reach zero, MAIN does two things: it shifts the display one to the right, and increments the time.

The display shifting has been discussed previously in section 2.2, and won't be reiterated here.

The time incrementing is performed in such a way that if the seconds overflow (i.e. they reach 60), then minutes will be incremented. If minutes overflows, hours will be incremented, etc. This could be done by writing explicit code for all three fields. This would be relatively verbose, however, and contain a lot of duplication. Consequently, the author took inspiration from the laboratory

examples and implemented the three time fields as consecutive memory, which can be iterated over in a generic fashion. While this doesn't particularly save much program memory in the assignment as it stands, it does scale quite well when adding more fields (e.g. days, months, years, etc), and makes doing so very trivial.

2.4 DISPLAY_TIME

The display time function is relatively complex, and consequently deserves a deeper investigation. To start with, it sets the LCD display cursor to the appropriate starting location, as determined by LCD_POSITION. It then erases four characters - this pads the display out by four characters, which helps centre the time on the display. It should be noted that this could also be done by simply adjusting the start position by 4, but this would not erase any existing value - which, when the time is scrolling, means the previous MSD of the hours, if not the whole time itself. It is just as easy to erase 4 characters as it is to skip 3 and erase 1, and it makes the code easier to understand in any case.

Having positioned the display cursor appropriately, the DISPLAY_TIME function then considers whether it needs to display the hours. If so, it [if necessary] calls the CONVERT_HOURS_TO_12 function to convert the hours from 24-hour notation to 12-hour. The CONVERT_HOURS_TO_12 uses a combination of logic, arithmetic and a table lookup to perform the conversion relatively efficiently, without resorting to manual BCD decrementing (which would require an extra function, in addition to the CONVERT_HOURS_TO_12 function). Once a suitable hours value is acquired, one way or another, it is displayed.

Once done, or if the hours were skipped, the minutes are displayed. Their display is quite straightforward. Similarly, seconds are then displayed (if necessary). Lastly, if 12-hour time is used the appropriate AM/PM designation is displayed. To determine which, the result previously returned from CONVERT_HOURS_TO_12 is compared to the actual HOURS value stored in memory. If they are the same, it must be AM; otherwise PM. This elegantly avoids having to perform a separate check, or return some additional flag from CONVERT_HOURS_TO_12.

It should be noted that both 12 and 24-hour modes start from 00:00, and run through to 11:59 or 23:59, respectively. This is technically accurate, although not what most people are used to - they would expect the zero hour to be displayed as 12. This could be done with a minimum of fuss, by substituting an hour of 00 with 12 or 24 (as appropriate) prior to display, but is pointless in the author's opinion.

Once the time is displayed, an additional 7 bytes are erased. This ensures that, when switching between modes of different lengths, any old data is appropriately erased. It could be skipped if the mode is known not to have changed (e.g. by recording such as a flag in some file), but it was felt the performance loss was imperceptible - and for our use irrelevant - and consequently worth the trade off in complexity.

3.0 Conclusion

The assignment was completed successfully, meeting and exceeding the specifications provided. Indeed, the author could have provided even more extra functionality, but felt a trade off between functionality (ergo complexity) and simplicity was necessary.

The assignment was exceptionally useful in learning how to use LCD displays, in addition to programming PIC devices. While embedded microcontrollers and microprocessors are numerous - and consequently learning to program any particular one may have little practical use - LCD displays are all very similar, and very commonly used. The author feels the experience has been invaluable.

4.0 Program Listing

```
; ASSIGN1.ASM
;
; Wade Tregaskis (02557793)
; Last modified 4th of May, 2005.


                                ; Registers

INDF:                   .EQU $00
OPTIONS:                .EQU $01        ; Only applies in bank 1.
PCL:                    .EQU $02
STATUS:                 .EQU $03
FSR:                    .EQU $04
PORTA:                  .EQU $05        ; Only applies in bank 0.
TRISA:                  .EQU $05        ; Only applies in bank 1.
PORTB:                  .EQU $06        ; Only applies in bank 0.
TRISB:                  .EQU $06        ; Only applies in bank 1.
INTCON:                 .EQU $0B


                                ; Bit masks and values

OPTIONS_DEFAULT:.EQU %00000101          ; Choose to use the pre-scaler for the timer, and set it to 1:64.
                                        ; This equates exactly 200 timer overflows with 1 second,
                                        ; given our 3.2768 MHz clock.

PAGEBIT:                .EQU 5          ; The bit of the OPTIONS file which controls what memory page we are in.

W:                      .EQU 0          ; Symbolic constant for the 'W' flag to various instructions.
F:                      .EQU 1          ; Symbolic constant for the 'F' flag to various instructions.

C:                      .EQU 0          ; The bit of the STATUS file representing arithmetic carry from the upper nibble.
DC:                     .EQU 1          ; The bit of the STATUS file representing arithmetic carry from the lower nibble.
Z:                      .EQU 2          ; The bit of the STATUS file representing a zero result.

LOWER_NIBBLE_MASK:      .EQU $0F        ; Used mainly for BCD to obtain only one of the two digits stored in a byte.
UPPER_NIBBLE_MASK:      .EQU $F0

TIMER_OVERFLOW:         .EQU 2          ; The bit in the INTCON file representing timer overflow.
ONE_SECOND_TIMER:       .EQU 80         ; In theory should be 200, but that's way too slow.
;ONE_SECOND_TIMER:      .EQU 1          ; Debug version.

LCD_POWER_ON_DELAY:     .EQU 40         ; The number of timer cycles to wait for the LCD to power up.
                                        ;   This amount of time will be allowed before the LCD is programmed,
                                        ;   and before it is sent it's first data (after being programmed).
LCD_RS:                 .EQU 4          ; The bit determining whether control or byte data is being sent to the LCD.
LCD_EN:                 .EQU 5          ; The bit who's falling edge is used to latch data into the LCD.
LCD_DELAY:              .EQU 20         ; The number of loops to perform after each write to the LCD.
                                        ;   This allows it some time to properly buffer it's input into memory.
LCD_WIDTH:              .EQU 40         ; The width of the LCD.. not likely to change, but just in case.

LCD_FIRST_ROW:          .EQU %10000000  ; The address of the first byte in the first row on the LCD.
LCD_SECOND_ROW:         .EQU %11000000  ; The address of the first byte in the second row on the LCD.

MODE_12:                .EQU 7          ; MSB of the TIME_SETTINGS byte
                                        ;   Determines whether to display in 12-hour (with AM/PM designation) or 24-hour format.
FORMAT_HM:              .EQU 0          ; HH:MM time format (in TIME_SETTINGS byte).
                                        ;   If set, display format is HH:MM.
FORMAT_MS:              .EQU 1          ; MM:SS time format (in TIME_SETTINGS byte).
                                        ;   If set, display format is MM.SS.

                                        ; HH:MM.SS format is implied by neither of these being set.
                                        ; If both are set, the result is undefined.


                                ; Macros

#DEFINE PAGE0   bcf STATUS, PAGEBIT     ; Changes the current memory page to page 0.
#DEFINE PAGE1   bsf STATUS, PAGEBIT     ; Changes the current memory page to page 1.


                                ; Variables

SECONDS:                .EQU $0C        ; Time values, as BCD digit pairs.
MINUTES:                .EQU $0D
HOURS:                  .EQU $0E
TIME_SETTINGS:          .EQU $1E        ; Settings such as display format and mode.


LCD_CONTROL:            .EQU $0F        ; Control data use on the LCD (presently only uses LCD_EN)

LCD_OUTPUT_TEMP:        .EQU $10        ; Used to hold W while working with the LCD output.
LCD_OUTPUT_TEMP2:       .EQU $11        ; Used to hold LCD_OUTPUT_TEMP while working with the LCD output (when doing wrapping).
LCD_WAIT:               .EQU $12        ; Holds the current wait count value for the LCD output delay loop.
LCD_ITERATOR:           .EQU $13        ; Holds the current position while iterating over a character string.

CLOCK_COUNT:            .EQU $14        ; Holds the current clock count - 'triggers' when reaches zero.
TIMER_COUNT:            .EQU $15        ; Holds the current timer count - used by TIMER_WAIT.

LCD_BCD_TEMP:           .EQU $16        ; Used to hold the original BCD value being displayed in LCD_OUTPUT_BCD.

INCREMENT_COUNT:        .EQU $17        ; The current iteration number of the time increment loop.

LCD_POSITION:           .EQU $18        ; The LCD offset from origin ($00 [and $40] being leftmost).
LCD_CURRENT_POSITION:   .EQU $19        ; The current LCD offset - used when printing the time to implement wrapping properly.
```

```
TIME_ERASE_COUNT:       .EQU $1A        ; The erase count when padding out the time to get rid of wrapping MSD of the hour.

PORTA_BUFFER:           .EQU $1B        ; The last value of PORTA read by the main loop (i.e. what buttons are currently down).
                                        ;   This is updated by GET_INPUT.  The result of GET_INPUT is a mask
                                        ;   of the buttons which have just been pressed - not those which were
                                        ;   already pressed last time GET_INPUT was called.

INPUT:                  .EQU $1D        ; Used to store the input, since we can't bit test W directly (booo).

TEMP:                   .EQU $20        ; Used by various functions for general store (not guaranteed to be unchanged after a 'call' or 're-
turn').


                                ; Program

        .ORG $0004                      ; Reset vector.
        .ORG $0005                      ; Program start.
        goto SETUP


TIME_LIMITS_TABLE:
        andlw %00000011                 ; Make sure the jump address is bounded.
        addwf PCL, F                    ; Jump into the table at the point specified by W.

        retlw $60                       ; Seconds.
        retlw $60                       ; Minutes.
        retlw $24                       ; Hours.
        retlw 0                         ; Placeholder - not used.


LCD_SETUP_TABLE:
        andlw %00000111                 ; Make sure the jump address is bounded.
        addwf PCL, F                    ; Jump into the table at the point specified by W.

        retlw %00110011                 ; Initialise the module (as recommended by "How to use Intelligent L.C.D.s").
        retlw %00110011                 ;   Repeated several times to make sure the LCD gets the messages,
        retlw %00110011                 ;   and initialises properly before we actually send real commands.
        retlw %00101000                 ; Configure the LCD for 4-bit input, 2 lines, 5x7 digits.
        retlw %00000110                 ; Set auto-increment on and auto display shift off.
        retlw %00001100                 ; Turn display on (no cursor or blink).
;       retlw %00001111                 ; Turn display on (cursor & blink) - debug version.
        retlw %00000001                 ; Clear display.
        retlw %00000010                 ; Reset all memory & stuff.


LCD_MESSAGE_LENGTH:             .EQU 26 ; The length of the message to display (in the table LCD_MESSAGE).
LCD_MESSAGE:
        andlw %00011111                 ; Make sure the jump address is bounded.
        addwf PCL, F                    ; Jump into the table at the point specified by W.

        retlw 'W'
        retlw 'h'
        retlw 'a'
        retlw 't'
        retlw $27                       ; "'", since TASM doesn't support character escaping.
        retlw 's'
        retlw ' '
        retlw 't'
        retlw 'h'
        retlw 'e'
        retlw ' '
        retlw 't'
        retlw 'i'
        retlw 'm'
        retlw 'e'
        retlw ','
        retlw ' '
        retlw 'M'
        retlw 'r'
        retlw '.'
        retlw ' '
        retlw 'W'
        retlw 'o'
        retlw 'l'
        retlw 'f'
        retlw '?'
        retlw ' '                       ; Unused - padded to prevent bad values in W causing unexpected jumps.
        retlw ' '
        retlw ' '
        retlw ' '
        retlw ' '
        retlw ' '

HOUR_CONVERSION_TABLE:
        andlw %00000011                 ; Make sure the jump address is bounded.
        addwf PCL, F                    ; Jump into the table at the point specified by W.

        retlw $08                       ; 20:00 -> 8:00pm.
        retlw $09                       ; 21:00 -> 9:00pm.
        retlw $10                       ; 22:00 -> 10:00pm.
        retlw $11                       ; 23:00 -> 11:00pm.


        ; LCD_OUTPUT_NIBBLE
        ;       Outputs the lower 4 bits of W to the LCD.

LCD_OUTPUT_NIBBLE:
        andlw LOWER_NIBBLE_MASK         ; Consider only the lower 4 bits of W.
        iorwf LCD_CONTROL, W            ; Include any control bits (e.g. LCD_RS).
```

```
        movwf PORTB                     ; Output the LCD data.
        bsf PORTB, LCD_EN               ; Setup the trigger.
        bcf PORTB, LCD_EN               ; Trigger the LCD (i.e. get it to latch the data we've provided).
        return


        ; LCD_OUTPUT_BYTE
        ;       Outputs an ASCII byte to the LCD.  This requires two calls to LCD_OUTPUT_NIBBLE,
        ;       for the high then the low nibble.  This does not modify the existing LCD_RS bit in
        ;       the LCD_CONTROL file; make sure this is set appropriately for your use beforehand.

LCD_OUTPUT_BYTE:
        movwf LCD_OUTPUT_TEMP           ; Save the current value.

        btfss LCD_CONTROL, LCD_RS       ; Don't think about wrapping if we're sending a command byte.
        goto LCD_OUTPUT_BYTE_START

        movwf LCD_OUTPUT_TEMP2          ; Save the current value again, since we're going to use LCD_OUTPUT_TEMP to send our control byte.

        movlw LCD_WIDTH                 ; Width of the display.
        subwf LCD_CURRENT_POSITION, W   ; Subtract the current position.
        btfss STATUS, Z                 ; Check if equal (i.e. we're at the end of the display, and need to wrap).
        goto LCD_OUTPUT_BYTE_INCSTART

        clrf LCD_CURRENT_POSITION       ; Reset the current position.
        movlw LCD_SECOND_ROW            ; Default wraps to the second row.
                                        ;   This means you can have one big long line starting from the first row,
                                        ;   or repeated wrapping on a single line in the second row.
        movwf LCD_OUTPUT_TEMP           ; Our control value to output.
        call LCD_OUTPUT_CONTROL         ; Recursion!
        movf LCD_OUTPUT_TEMP2, W        ; Load our original character to display.
        movwf LCD_OUTPUT_TEMP           ; Restore the original character to display.

LCD_OUTPUT_BYTE_INCSTART:
        incf LCD_CURRENT_POSITION, F    ; Increment the remembered LCD position for each data byte outputted.

LCD_OUTPUT_BYTE_START:
        movlw LCD_DELAY                 ; Delay value - experimentation shows this works well enough.
        movwf LCD_WAIT
LCD_OUTPUT_BYTE_DELAY:
        decfsz LCD_WAIT, F              ; Decrement delay counter, exit once expired.
        goto LCD_OUTPUT_BYTE_DELAY
        ; Exit loop

        swapf LCD_OUTPUT_TEMP, W        ; Load in the current value nibble-swapped.
        call LCD_OUTPUT_NIBBLE          ; Output the high nibble.
        movf LCD_OUTPUT_TEMP, W         ; Load in the current value.
        call LCD_OUTPUT_NIBBLE          ; Output the low nibble.
        return


        ; LCD_OUTPUT_CONTROL
        ;       Outputs a control byte (passed in W) to the LCD - setting LCD_RS automatically beforehand,
        ;       and then resetting LCD_RS afterwards.

LCD_OUTPUT_CONTROL:
        bcf LCD_CONTROL, LCD_RS         ; Clear RS to indicate this is a control byte.
        call LCD_OUTPUT_BYTE            ; Output the control byte.
        bsf LCD_CONTROL, LCD_RS         ; Set RS on the assumption that normal operation uses data bytes.
        return


        ; LCD_OUTPUT_BCD
        ;       Outputs a BCD number, where the high nibble of the byte in W is considered the first digit,
        ;       and the low nibble the second.

LCD_OUTPUT_BCD:
        movwf LCD_BCD_TEMP              ; Save the value to be restored later.
        call LCD_OUTPUT_BCD_CORE        ; This is a neat little trick to perform a single recursion.
                                        ;   Inspired by the code from lab 29/30/etc.

LCD_OUTPUT_BCD_CORE:
        swapf LCD_BCD_TEMP, F           ; Swap the data (on first run this will put the real high nibble into the low nibble.
                                        ;       The next time through it will reverse this, so we get back to the real low
                                        ;       nibble in the low nibble).
        movf LCD_BCD_TEMP, W            ; Load the data.
        andlw LOWER_NIBBLE_MASK         ; Mask out the high nibble, since we want only one BCD digit at a time.
        addlw '0'                       ; Add ASCII zero to the nibble, to determine the ASCII value of the decimal number.
        call LCD_OUTPUT_BYTE            ; Output the ASCII number of the current digit.
        return                          ; Return - the first time this actually returns to LCD_OUTPUT_BCD_CORE, so as to output
                                        ;       the low nibble, having done the high.  The second time it does the full return
                                        ;       to the caller of LCD_OUTPUT_BCD.


        ; TIMER_WAIT
        ;       Waits for a number of timer cycles to expire, as determined by the value passed in W.

TIMER_WAIT:
        movwf TIMER_COUNT               ; Setup the initial loop counter.
TIMER_WAIT_LOOP:
        btfss INTCON, TIMER_OVERFLOW    ; Test for timer overflow.
        goto TIMER_WAIT_LOOP            ; Loop if no overflow detected.
        bcf INTCON, TIMER_OVERFLOW      ; Reset timer overflow bit.
        decfsz TIMER_COUNT, F           ; Test if our timer tick count has expired.
        goto TIMER_WAIT_LOOP            ; Repeat if not.
        return                          ; Otherwise return to caller.


        ; DISPLAY_MESSAGE
        ;       Displays 'the' message.  This will take some time, as the message can be quite long.
```

```
        ;       This is usually called only once, as the message is never modified.
        ;       This will automatically set the LCD position to the first byte of the first row, and go from there.
        ;       It will not restore any such LCD state on return - make sure to re-setup the LCD as required.

DISPLAY_MESSAGE:
        clrf LCD_CURRENT_POSITION       ; We always start at $00... this must correspond to the first byte of the first row on the LCD.

        movlw LCD_FIRST_ROW             ; Select the first row of the LCD to display on.
        call LCD_OUTPUT_CONTROL         ; Send command.

        clrf LCD_ITERATOR               ; Reset the iterator to the start of the message.
DISPLAY_MESSAGE_LOOP:
        movf LCD_ITERATOR, W            ; Load the current iterator value into W.
        call LCD_MESSAGE                ; Get the current message byte.
        call LCD_OUTPUT_BYTE            ; Output the current byte.

        incf LCD_ITERATOR, F            ; Increment the iterator.
        movf LCD_ITERATOR, W            ; Load the new iterator value.
        xorlw LCD_MESSAGE_LENGTH        ; Compare with the message length, to determine if we're at the end.
        btfss STATUS, Z                 ; Exit the loop if at end of message.
        goto DISPLAY_MESSAGE_LOOP       ; Otherwise repeat, displaying the next character in sequence.
        return                          ; Exit loop and return to caller.


        ; DISPLAY_ERASE
        ;       Erases some number of characters from the display (determined by the value passed in in W),
        ;       starting at the cursor and auto-incrementing.  'Erasing' means writing over with a space.
        ;
        ;       Danger Will Robinson, Danger!  This will not check for wrapping, so be very careful.

DISPLAY_ERASE:
        movwf TIME_ERASE_COUNT          ; Our argument in W determines how many characters to erase.
DISPLAY_ERASE_LOOP:
        movlw ' '                       ; Load our erase character, i.e. a space.
        call LCD_OUTPUT_BYTE            ; Spit it out to the LCD.
        decfsz TIME_ERASE_COUNT, F      ; Decrement our count and check if it's expired.
        goto DISPLAY_ERASE_LOOP         ; Loop if it hasn't.
        return                          ; Otherwise return to caller.


        ; CONVERT_HOURS_TO_12
        ;       Converts the true 24-hour value stored internally to a form suitable for 12-hour display.
        ;       You can then determine whether the given time is AM or PM by comparing the result of this
        ;       function with the original value - if it has changed, the time is in PM.
        ;
        ;       Note that the result of calling this function with BCD values outside 00 to 23 is undefined.

CONVERT_HOURS_TO_12:
        ; if (MSD bit 0 is set) then // MSD == 1
        ;       if (LSD bits 1, 2 or 3 are set) then // LSD > 1
        ;               subtract 8 from LSD
        ;               zero MSD
        ;       end if
        ; else if (MSD bit 1 is set) then // MSD == 2
        ;       zero MSD
        ;       call lookup
        ; end if

        btfss HOURS, 4                  ; Check if hours is 1X.
        goto CONVERT_HOURS_TO_12_NEXT   ; If it isn't go to the next check.

        movlw %00001110                 ; Check if LSD is > 1 (i.e. HOURS is >= 12)
        andwf HOURS, W

        btfss STATUS, Z
        goto CONVERT_HOURS_TO_12_SUB2   ; HOURS is >= 12, so convert it to 12-hour form.

        movf HOURS, W                   ; Reload original value, since is an AM time and thus needs no modification.
        return                          ; Return original value to caller.

CONVERT_HOURS_TO_12_SUB2:
        movlw 2                         ; LSD is 2 through 9, so need to reduce and zero MSD.
        subwf HOURS, W                  ; Reduce LSD by 2; i.e. 19 becomes 17, which when the MSD is wiped becomes 7, which is correct.
        andlw LOWER_NIBBLE_MASK         ; Zero MSD.
        return

CONVERT_HOURS_TO_12_NEXT:
        btfss HOURS, 5                  ; Check if hours is 2X (e.g. 20 through 23).
        return                          ; It it isn't, implying we must be 0X and thus already in suitable form, exit.

        movf HOURS, W                   ; Load the original HOURS value.
        andlw LOWER_NIBBLE_MASK         ; Mask out the high digit.
        call HOUR_CONVERSION_TABLE      ; Call our magic table to convert these nasty four values.
        return


        ; DISPLAY_TIME
        ;       Displays the time.  This should be called whenever the time is updated.
        ;       It handles all formatting and related concerns (although it does expect the LCD to be
        ;               already initialised for display).

DISPLAY_TIME:
        movf LCD_POSITION, W            ; Copy the start position to the current position.
        movwf LCD_CURRENT_POSITION

        movlw LCD_SECOND_ROW            ; Select the second row of the LCD to display on.
        addwf LCD_POSITION, W           ; Adjust with the current offset.
        call LCD_OUTPUT_CONTROL         ; Send command.

        movlw 4                         ; Erase the leading 4 bytes so we're centered properly.
```

```
        call DISPLAY_ERASE

        btfsc TIME_SETTINGS, FORMAT_MS  ; Check if we're in MM:SS mode, in which case we skip the hours.
        goto DISPLAY_TIME_MINUTES


        movf HOURS, W                   ; Display hours.

        btfss TIME_SETTINGS, MODE_12    ; Test our time settings to see if we're in 12 hour mode).
        goto DISPLAY_TIME_HOURS         ; If not, just display the time as-is.

        call CONVERT_HOURS_TO_12        ; Convert our 24-hour form to 12-hour form.
        movwf TEMP                      ; And store for later comparison in TEMP.
        ; Fall through to display hours

DISPLAY_TIME_HOURS:
        call LCD_OUTPUT_BCD
        movlw ':'                       ; Display hh:mm separator, ":".
        call LCD_OUTPUT_BYTE

DISPLAY_TIME_MINUTES:
        movf MINUTES, W                 ; Display minutes.
        call LCD_OUTPUT_BCD

        btfsc TIME_SETTINGS, FORMAT_HM  ; Check if we're using HH:MM format, in which case we skip seconds.
        goto DISPLAY_TIME_SKIP_SECONDS

        movlw '.'                       ; Display mm:ss separator, ".".
        call LCD_OUTPUT_BYTE
        movf SECONDS, W                 ; Display seconds.
        call LCD_OUTPUT_BCD

DISPLAY_TIME_SKIP_SECONDS:
        btfss TIME_SETTINGS, MODE_12    ; Check if we're using 12-hour mode.
        goto DISPLAY_TIME_FINISH        ; If not, skip this AM/PM stuff.

        movlw ' '                       ; Display space.
        call LCD_OUTPUT_BYTE

        movf TEMP, W                    ; Compare the original hours with the display version to detect differences, which if present imply PM
time.
        xorwf HOURS, W
        btfss STATUS, Z                 ; Check if it's going to be an 'A' or a 'P'.
        goto DISPLAY_TIME_SHOW_P

        movlw 'A'
        goto DISPLAY_TIME_SHOW_A_OR_P

DISPLAY_TIME_SHOW_P:
        movlw 'P'
        ; Fall through

DISPLAY_TIME_SHOW_A_OR_P:
        call LCD_OUTPUT_BYTE            ; Display 'A' or 'P'.
        movlw 'M'                       ; Display 'M'.
        call LCD_OUTPUT_BYTE


DISPLAY_TIME_FINISH:
        movlw 7
        call DISPLAY_ERASE              ; Erase the trailing 7 bytes to avoid wrapping the MSD of the hours.
                                        ;   Note that we don't always need to erase 7 bytes - just one if we
                                        ;   use HH:MM:SS in 12-hour form.  But, for simplicity, 7 covers all bases.
        return


        ; BCD_INC
        ;       Increments a BCD value (stored as two BCD digits in a single byte).
        ;       The value to increment is *not* stored in W, but rather should be addressed via INDF,
        ;       i.e. the address placed in FSR.

BCD_INC:
        incf INDF, F                    ; Increment the value to start with.
        movlw LOWER_NIBBLE_MASK         ; Load up our nibble mask.
        andwf INDF, W                   ; Mask our value, the one we just incremented.
        xorlw $0A                       ; Compare it to decimal 10 to see if it overflowed.
        btfss STATUS, Z
        return                          ; If not, return, since we're now done.
        movf INDF, W                    ; Otherwise load up the value, since we'll need to reset the low digit and increment the high digit.
        andlw $F0                       ; Mask out the LSD first, to increment the MSD.
        addlw $10                       ; Which we do here... it's easier to add $10 than swap the nibbles and incf.
        movwf INDF                      ; Save our incremented result to the original file.
        return


        ; GET_INPUT
        ;       Checks for new inputs on the five push-buttons connected to port A.
        ;       This function updates the PORTA_BUFFER variable with the status of port A as read upon invocation.
        ;       It returns in W a bit map of which buttons have been newly pressed, since the last invocation of GET_INPUT.
        ;       It does not magically catch any press that occurs - only those it observes while invoked.  Consequently,
        ;               it should be called regularly.

GET_INPUT:
        movf PORTA, W                   ; Load PORTA...
        movwf TEMP                      ; ...and save it in our temp file.

        xorwf PORTA_BUFFER, W           ; Determine which bits have changed (marking changed bits in W).
        xorwf PORTA_BUFFER, F           ; Store the original W value (i.e. PORTA) into PORTA_BUFFER for use next time we call GET_INPUT.
        andwf TEMP, W                   ; Mask out any bits which have turned off (i.e. we detect rising edge, not falling).
```

```
        return                          ; Return with the result in W.


        ; MANUAL_INC
        ;       Increments a given field (determined by the index passed in W, 0 being SECONDS, 1 MINUTES, 2 HOURS).
        ;       Fields will wrap appropriately, but do not auto-increment the next field when doing so.

MANUAL_INC:
        movwf TEMP                      ; Save the argument for use later when calling TIME_LIMITS_TABLE.

        movwf FSR                       ; Move the argument (W) into FSR.
        movlw SECONDS                   ; Load up our base, SECONDS.
        addwf FSR, F                    ; Add it to FSR to get the right field (SECONDS, MINUTES or HOURS).
        call BCD_INC                    ; Increment the field.

        movf TEMP, W                    ; Restore the original argument.
        call TIME_LIMITS_TABLE          ; Find the limit value for the current field.
        xorwf INDF, W                   ; Check if it matches the current value.
        btfsc STATUS, Z
        clrf INDF                       ; If it does, clear the current field, since it has overflowed.
        return


        ; SETUP
        ;       Called on startup or reset.  Configures the timers, LCD, etc, then falls into main,
        ;       where the program loops indefinately.

SETUP:
                                ; 1) Configure hardware

        PAGE0                           ; Just to be sure we're in page 0.
        clrf PORTA                      ; Clear latches to prevent any unusual behaviour.
        clrf PORTB

        PAGE1                           ; Configure I/O ports & timer.

        movlw %00011111                 ; Configure port A for input.
        movwf TRISA
        clrf TRISB                      ; Configure port B for output (LCD control etc).

        movlw OPTIONS_DEFAULT           ; Load up our default options.
        movwf OPTIONS                   ; Apply our default options.

        clrf INTCON                     ; Disable all interrupts and clear timer interrupt flag.

        PAGE0                           ; Return to page 0 for normal programming.


                                ; 2) Set initial values

        clrf HOURS                      ; Reset the time to midnight.
        clrf MINUTES
        clrf SECONDS
        clrf TIME_SETTINGS              ; Revert to generic default settings.

        clrf LCD_POSITION               ; We reset the display to home in the config loop, so we can safely reset the position here.
        clrf LCD_CONTROL                ; Clear the RS bit so everything's treated as control bytes.


                                ; 3) Configure LCD

        movlw LCD_POWER_ON_DELAY        ; Load up our power-on delay timer count.
        call TIMER_WAIT                 ; Wait for some time for the LCD to power on properly.

        clrf LCD_ITERATOR               ; Reset the iterator.
LCD_CONFIG_LOOP:
        movf LCD_ITERATOR, W            ; Load the current iterator value into W.
        call LCD_SETUP_TABLE            ; Get the current setup byte.
        call LCD_OUTPUT_BYTE            ; Output the current byte.

        incf LCD_ITERATOR, F            ; Increment the iterator.
        btfss LCD_ITERATOR, 3           ; Exit the loop if >= 8.
        goto LCD_CONFIG_LOOP

        movlw LCD_POWER_ON_DELAY        ; Give the LCD plenty of time to settle.
        call TIMER_WAIT                 ; Wait once again for the LCD.

        call DISPLAY_MESSAGE            ; Display the message.
        call DISPLAY_TIME               ; Display the initial time.
        ; Fall through to main


                                ; 4) Wait for input and do clock timing

        ; MAIN
        ;       Checks every loop for:
        ;               a) Button presses
        ;               b) Timer overflows
        ;       It handles the former appropriately, updating the time immediately to reflect any changes.
        ;       The latter cause CLOCK_COUNT to be decremented.  Every time it reaches zero, one second
        ;               is deemed to have passed, and the time is automatically incremented by one second.
        ;
        ;       MAIN never ends - it loops indefinately.  It should thus never be 'call'ed, only 'goto'd.

MAIN:
        movlw ONE_SECOND_TIMER          ; Load our one second timer count.
        movwf CLOCK_COUNT               ; Save it in our clock count file.

MAIN_LOOP:
                        ; Check inputs
```

```
        call GET_INPUT                  ; Check for new button presses.
        btfsc STATUS, Z                 ; If there are none, skip this input stuff.
        goto MAIN_LOOP_CHECK_TIMER

        ; We have input
        movwf INPUT                     ; Save the input so we can bit test against it.

MAIN_LOOP_CHECK_SECONDS:
        btfss INPUT, 0                  ; Test for seconds increment button.
        goto MAIN_LOOP_CHECK_MINUTES

        movlw 0
        call MANUAL_INC

MAIN_LOOP_CHECK_MINUTES:
        btfss INPUT, 1                  ; Test for minutes increment button.
        goto MAIN_LOOP_CHECK_HOURS

        movlw 1
        call MANUAL_INC

MAIN_LOOP_CHECK_HOURS:
        btfss INPUT, 2                  ; Test for hours increment button.
        goto MAIN_LOOP_CHECK_FORMAT

        movlw 2
        call MANUAL_INC

MAIN_LOOP_CHECK_FORMAT:
        btfss INPUT, 3                  ; Test for format button (hm/ms/hms).
        goto MAIN_LOOP_CHECK_MODE

        incf TIME_SETTINGS, F           ; Change the format.  In the case we increment to 3, we need to detect that and wrap to zero.
        btfss TIME_SETTINGS, FORMAT_HM
        goto MAIN_LOOP_CHECK_MODE
        btfss TIME_SETTINGS, FORMAT_MS
        goto MAIN_LOOP_CHECK_MODE

        bcf TIME_SETTINGS, FORMAT_HM    ; And this is the "wrapping to zero" bit... there's no single command to clear two bits, unfortu-
nately.
        bcf TIME_SETTINGS, FORMAT_MS

MAIN_LOOP_CHECK_MODE:
        btfss INPUT, 4                  ; Test for the 12/24-hour display mode button.
        goto MAIN_LOOP_FORCE_UPDATE

        movlw (1 << MODE_12)            ; Toggle the mode bit of the TIME_SETTINGS byte.
        xorwf TIME_SETTINGS, F

MAIN_LOOP_FORCE_UPDATE:
        call DISPLAY_TIME               ; Force an update of the display since we've changed the time.

MAIN_LOOP_CHECK_TIMER:
        btfss INTCON, TIMER_OVERFLOW    ; Check for timer overflow.
        goto MAIN_LOOP
        bcf INTCON, TIMER_OVERFLOW      ; Reset the timer.

        decfsz CLOCK_COUNT, F           ; Decrement our timer count and, if it has expired, exit the loop to update the time.
        goto MAIN_LOOP

                        ; Shift display

        movlw %00011000                 ; Command to shift the display right by one digit.
        call LCD_OUTPUT_CONTROL         ; Send the command.

        incf LCD_POSITION, F            ; Increment the position to match the display.
        movlw LCD_WIDTH
        subwf LCD_POSITION, W           ; Compare our current position with the maximum.
        btfsc STATUS, Z                 ; Skip the clear if we're not at the maximum (40).
        clrf LCD_POSITION

                        ; Increment time

        movlw SECONDS                   ; Load up the address of SECONDS.
        movwf FSR                       ; Store it in FSR.
        clrf INCREMENT_COUNT            ; This refers to the number of fields we have incremented,
                                        ;   i.e. if 0 we will next increment SECONDS, if 1 we will next increment MINUTES, etc.

MAIN_LOOP_INCREMENT_LOOP:
        call BCD_INC                    ; Increment the current field.
        movf INCREMENT_COUNT, W         ; Load our increment count, which determines which field we are working with.
        call TIME_LIMITS_TABLE          ; Retrieve the wrap value for the given field.
        subwf INDF, W                   ; Subtract the current field value from the wrap value.
        btfss STATUS, Z                 ; Test if the same.
        goto MAIN_END                   ; If not, skip to end, since we have no more work to do.

        clrf INDF                       ; Otherwise, clear the current field and see if we should increment any further fields.
        incf FSR, F                     ; So change FSR to point to the next field.
        incf INCREMENT_COUNT, F         ; Increment our field count (i.e. field pointer).
        movlw 3                         ; We want to compare it against 3, which is the sentinal value for our field pointer.
        xorwf INCREMENT_COUNT, W        ; Compare the two...
        btfss STATUS, Z
        goto MAIN_LOOP_INCREMENT_LOOP   ; If we haven't gone past the last field, increment it.
        ; Fall through to MAIN_END      ; Else fall through to the end of MAIN.


MAIN_END:
        call DISPLAY_TIME               ; Display the new time.
        goto MAIN                       ; Repeat the whole sh!.
.END
```