MAZI ALARM
Developer's Guide

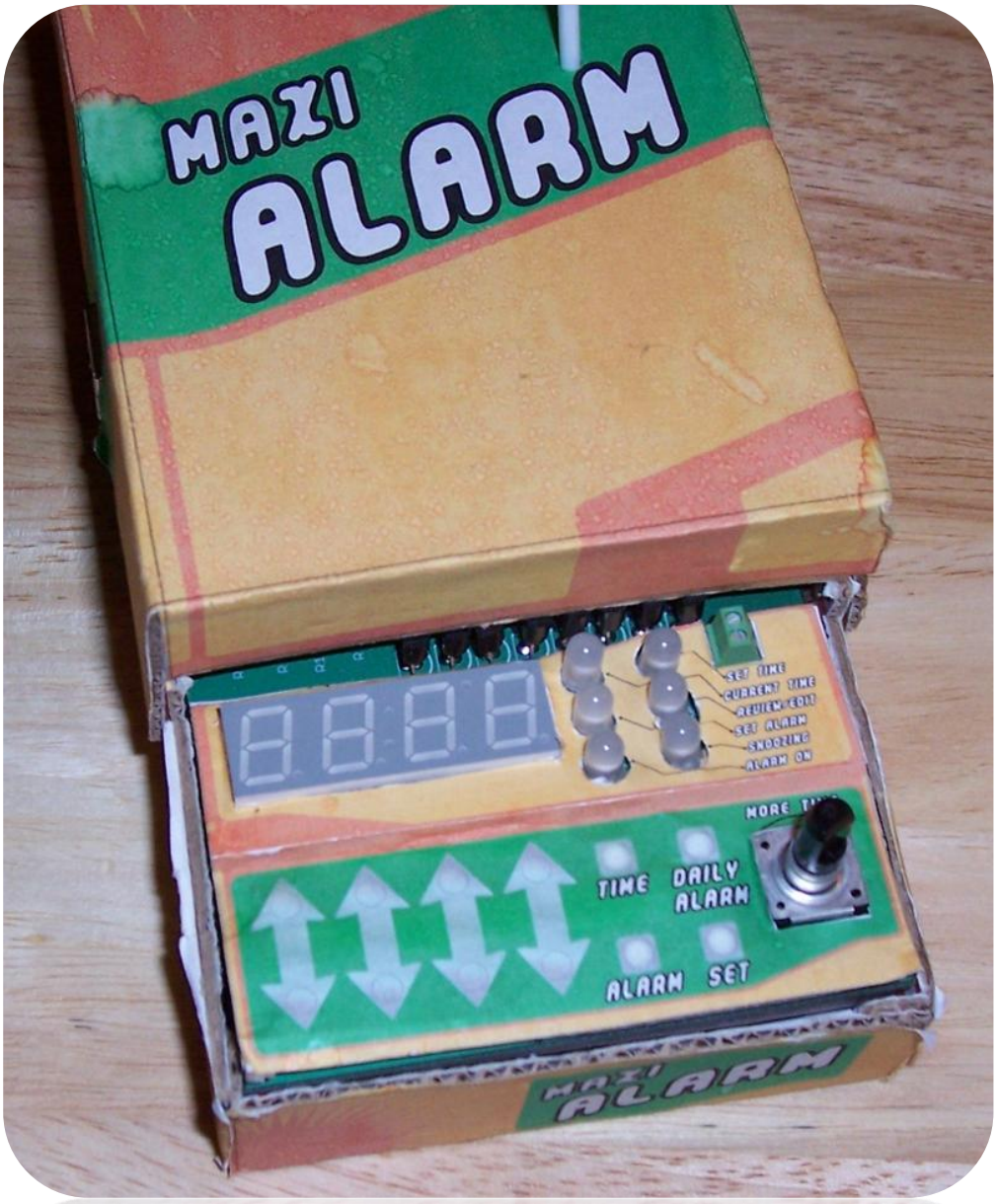


Table of Contents

About this Manual

Overview of this Manual's Layout

This manual will start off with general explanations, which will then lead into a deeper discussion of each structure and will end with appendices. These list certain aspects of the program separate from the overall picture. After the general description of the alarm clock, functions/subroutines will be grouped together and described based on how each system acts on as whole. Please be familiar with and refer to Microchip's document No. 39632D, PIC18F2455/2550/4455/4550 Data Sheet, available from their website: http://www.microchip.com/. It will be assumed you, the developer, will find the basic programming techniques listed in that manual. What will is not covered in that manual, and will be talked about briefly, is the P24 hardware with which PIC18 interfaces.

Notations and Convention

See Appendix A: Acronyms and Abbreviations.

Number formats

Numbers will be listed as "0x" for hexadecimal interpretation and without "0x" for decimal interpretation. They will be stored Little Endian style, the lowest byte at the lowest address. Registers hold 8 bits of data, instructions take up 2 Bytes in program memory, and the program counter can only point to every other address in program memory.

1 Word = 2 bytes

Programming Consistencies

Assembler directives will be in CAPITAL LETTERS.

Tabs

Please view with you text editor set for tabs being displayed as 4 spaces. Function labels will exist without being tabbed over; labels within functions will be preceded by two spaces. Instructions will be preceded by one tab, instruction arguments three tabs (from beginning of the line).

Comments

When explaining a section of code, they will be lined up with the instructions (one tab). When to the right of an instruction, they will explain that instruction's significance.

Not as constant: ";;" starts an idea, followed on the next line(s) by ";" then ideas end with ";;" For example:

```
;; if here, the user has dialed the snooze down to zero,
;     Turn off snooze alarm. (change the day back 1, just to
;     make sure it doesn't ring.
CLRF  MASTER_STATE, 0                   ;; Turn off snooze alarm, set state
CLRF  DelayState, 0                     ;     since we've used it for RE
DECF  INDF0 , 1, 0                      ;     points to SA[date[low]] from before
GOTO  IdentidemVicis                    ;;    Exit snooze stuff.
```

Buttons

When you see 'x' related to a button (pressing 'x', 'x' button) that character refers to a button as such:

'a' = ALARM

'd' = DAILY ALARM

's' = SET

't' = TIME

\/\ = Arrow buttons

Other Notations

Function names in the form QdescriptiveNameQ usually behave as if answering a question, where the question is related to the descriptive name.

Un-updated References

I am sure you, the reader, will notice many page cross references, but also many pages referenced as $$. I did not have time to update all of these. So when you see a $$, it refers to that item's entry in the state list, function list, variable or macro lists in the appendices (I apologize if you cannot tell the difference, this will make it harder to find it, try searching the name).

Overview of the System

General Description

First we must note that the first 0x1000 lines of code on the PIC are reserved for the boot program, thus everything for us starts at address 0x1000 and up.

On reset the main function sets up data and interrupts for other functions to run. After this we enter the main loop, *IdentidemVicis,* (for responsibilities of the main loop, see page 8). The program flow, as a whole, is directed by which state is currently being occupied (list of states on page 24). Tasks in the main loop and interrupts function as ways to update data for each state, with a few exceptions.

Figure 1 is the front panel of the display, it dictates which buttons do what functions and which LED's have certain significances.

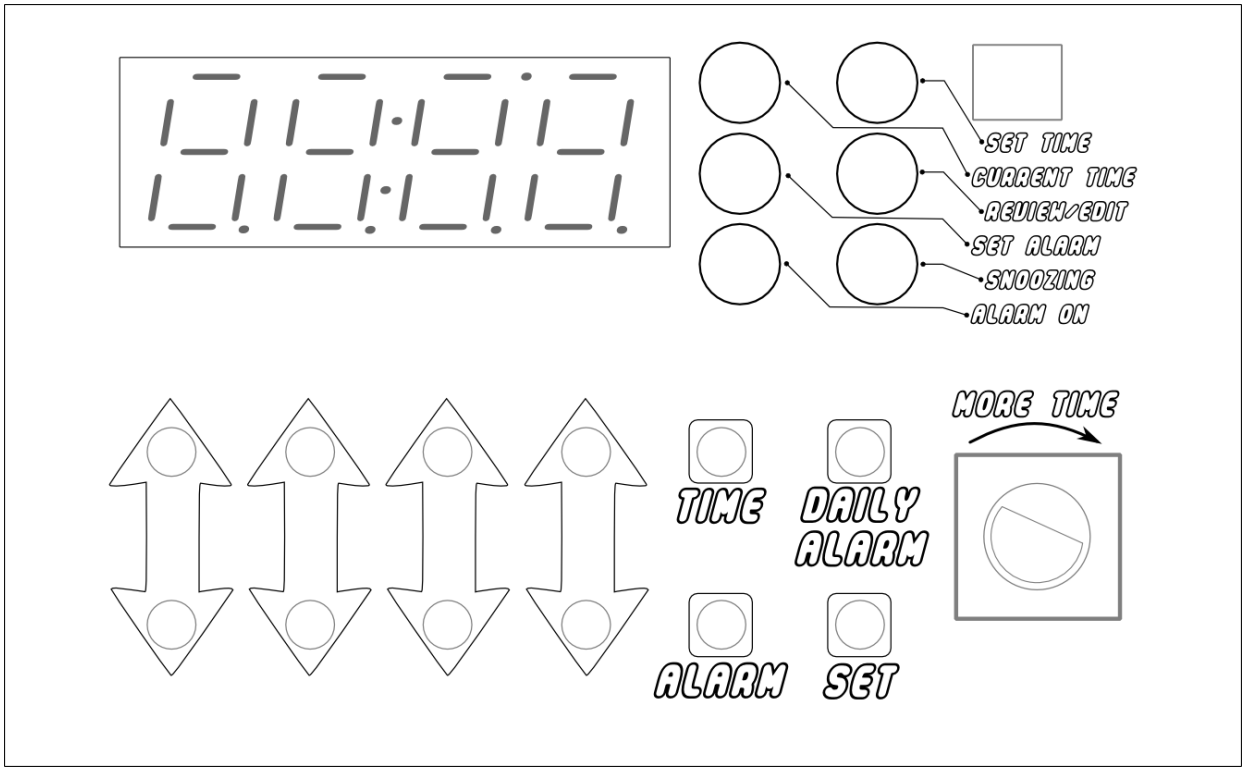


Figure 2

Switches are arranged as such:

1	3	5	7	9	B	(1-8 are arrows, 9-C are mode buttons)
2	4	6	8	A	C	

LED's are labeled according to their matrix value: (column, row) starting in the upper left corner.

Program Flow

Reset

- Initiate registers and interrupts (page 9).

Main Loop

- Refresh LED's (page 12).
- See if alarm needs to start ringing (page 22).
- See if we are snoozing, execute snoozing code if that is the case (page 22).
- See if we need to change states based on our delay counter (page 23).
- Go to the current state's code (page 23).
- Repeat

Interrupts

- Update the time/date, decrement time dependent counters (page 11).
- Update the switch state registers (page 12).
- Update the rotary encoder (page 17).
- Sound alarm, not implemented.

States

- Everything else (see all states on page 24).
- Fill display buffer (page 12).
- Set up delay state system (page 17).
- Respond to user input (page 15).
- Update every other global status register not done by the main loop or interrupts.

Description of Source Files

These files appear as they have when I wrote them. Comments were made at the time, so they are genuine, but may be a little disorganized. The code has not been cleaned up either (but should match the structure I outlined on page 5 for the most part).

P18F4550.INC

Contains the normal include information to compile assembly files for the PIC18F4550.

18F4550.lkr

Modified for 0x1000 start, use this one, not the default linker file.

ENEE 440 - SimOnly.asm

Include with making project for simulation purposes (when program counter comes to address 0x0, 0x10, 0x18 the get redirected to where address 0x1000, 0x1010 and 0x1018 direct to (respectfully).

ENEE 440 - PIC18 Alarm Clock Alarms.asm

Holds alarm constants and functions to set, alter, view alarms and the alarm stacks (page 19).

ENEE 440 - PIC18 Alarm Clock Buzz.asm

Not used, would make sure sounds happens when called from interrupt, but not completed yet.

ENEE 440 - PIC18 Alarm Clock High Int.asm

Directs High interrupt (page 9).

ENEE 440 - PIC18 Alarm Clock LED.asm

Contains constants and functions to refresh display, JSCII constant values as well as output hardware constancies (page 12).

ENEE 440 - PIC18 Alarm Clock Low Int.asm

Manages low priority interrupts with sub priorities (page 9).

ENEE 440 - PIC18 Alarm Clock Main.asm

Main.asm contains the main function, interrupt declaration, calling of resets, main loop, managing of states, alarms and snoozing status (see page 22). Also contains code for each state (see page 24).

ENEE 440 - PIC18 Alarm Clock MT.asm

Controls master time, called from by high priority interrupt (see page 11). Also updates other time sensitive counters (see pages 15, 17).

ENEE 440 - PIC18 Alarm Clock RE.asm

Keeps track of rotary encoder, on interrupt (see page 17).

ENEE 440 - PIC18 Alarm Clock SW.asm

Contains switch constants and functions updated on interrupt (see page 13).

Interrupts

High Interrupt

The high priority interrupt is responsible for keeping track of time. It fires when Timer 1's counter overflows. This interrupt fires every hundredth of a second (see page 11 for how this is accomplished). Every hundredth of a second MASTER_TIME (page $$) gets incremented once.

Low Interrupt (with dispatcher)

The low priority interrupt is divided into two sub-priority interrupts in this implementation. Up to eight sub-priority interrupts can be managed with the two macros: Sub_P_set_L and Sub_P_ex_L, each interrupt executing both. It may be helpful to be looking at "ENEE 440 – PIC18 Alarm Low Int.asm" now.

The former macro (executed first) looks to see if its interrupt has fired, if so update Start_Srvs (see page $$) to acknowledge this (states that we need to start servicing this interrupt). Then we clear that interrupt's flag so we do not interrupt again.

The latter macro looks to see if we are in the middle of an interrupt, if so, resume. IMPORTANT: if this is the case, a lower sub-priority interrupt has interrupted a higher sub-priority interrupt, so we wish to continue with the higher sub-priority interrupt, which is why the macros should be written with the highest sub-priority first. (Doing these out of order changes which bits correspond to the higher priority). If this current interrupt in not in progress, we check to see if it needs to be started (looking at bit in Start_Srvs). If so start it (if two interrupts occur at the same time, this will cause the higher priority to start before the other one does), along with starting to service, we clear its bit in Start_Srvs and set its bit in In_Service_L (see page $$) which tells us it is running.

Latency

The latency to respond to an interrupt grows with each sub-priority we implement. The highest one will start in *4·n + 6 cycles*, where n is the number of sub-priority interrupts. This assumes each interrupt fires at once. Note that in by now we have already taken care of the interrupt flags, the only thing we have not done is backup any variables we need to save. The lowest sub-priority will start in 3·n + 5·n = 8·n cycles. This assumes that no other interrupts have occurred/still being worked on. The time it takes to resume a higher sub-priority interrupt when a lower sub-priority interrupt fires is about 4·n + 3 cycles.

Tasks associated with sub-priorities

The switches are updated at 22.9 Hz via timer 0. (lower sub-priority: 1)

The rotary encoder position is updated at 6510 Hz via timer 2. (higher sub-priority: 3)

The buzzing sound: updated at 3520 Hz, but the interrupt is not actually set up, so it will never ring (actually just adds latency right now, but kept in since we will want our alarm to make noise, may not be needed with Pulse Width Modulator). (highest 'implemented' sub-priority: 6)

Modes

The word 'Modes' will refer to systems of states and interrupts acting together to serve an overall independent idea.

Keeping time

Accuracy

Off by 12 minutes and 32 seconds per year

Storing time

Aside for what is done in the states, this is where most of the work happens in this alarm clock. Time is kept in MASTER_TIME (page $$), MASTER_DATE (page $$) and MASTER_DAY (page $$), which hold the current time (hours, minutes, seconds, hundredth of seconds), current date (0000 through 9999) and current day of the week. The first two use compact binary coded decimal (compact BCD) to store their numbers.

- Hundredth of seconds range from 00 to 99 in the lowest byte of MASTER_TIME.
- Seconds range from 00 to 59 in the second lowest byte of MASTER_TIME.
- Minutes range from 00 to 59 in the third lowest byte of MASTER_TIME.

- Hours range from 01 to 24 in the highest byte of MASTER_TIME.
- Date ranges from 0000 to 9999 in both bytes of MASTER_DATE.
- Day of the week is one bit that gets rotated in the 8 bits of MASTER_DAY (-1 is not allowed).

Interrupts, timing and latency

The frequency which we update the time is 1/100 of a second. This is accomplished via the high interrupt. WREG, STATUS and BSR registers are stored automatically, and we get straight access to this code with minimal latency. The interrupt fires when timer 1's counter overflows, this counter is a 2^n number. Our clock runs at 48 MHz, and 48 MHz $\cdot 2^n \neq 100$ Hz for any n. So how do we get it to interrupt at 100 Hz: adjust the timer's counter. At 48 MHz we divide by 4 (four cycles/instruction) to get then by 2 (for prescale) to get the number of instructions executed in 0.01 seconds: 60,0000 instructions. So if we could set our counter to -60,000 we would overflow every 0.01 seconds. This can be done by manually setting our counter to 15A0 (since -60,000 = 0x…FF15A0). This would be wonderful if it could be done at the time of overflow, but it cannot. So we must account for latency, by setting our counter to -60,000 + latency. This latency was found experimentally via the simulator but there are 6 instructions that need to happen before our counter gets reset, so set counter to 0x15A6.

Error in the time explained: If high interrupt latency was constant (it is close) we would have not time lost/error in our system. But that is not the case. The latency is sometimes more that 6 instructions, which introduces error. Experimentally (see README.txt) we have a relative error of 0.000023826 = 0.0023826%, or loosing 12 minutes and 32 seconds per year, or 14.4 seconds per week.

Other time keeping tasks

We take care of some other tasks in the time keeping interrupt, decrementing the state delay counter (page 17), decrementing the switch hold down counter (page 15) flashing the colon at 1 Hz, 5 Hz or off based on the current state, and if the alarm in ringing: flash the LED's red, yellow, green, off at 1 Hz (for display information, see page 12).

Interrupt Justification

Normally one would not expect an interrupt to commit this much time from the normal program, but here are some reasons why I decided to implement all of these tasks in the interrupt:

1) Increment the time, takes time to calculate hundredths of seconds up to days of the week.
   Want this to be done as soon as possible, could set another flag and handle it the main loop, but that would take more time, and replicate handling the interrupt flag outside an interrupt.
2) Decrement timers.
   If updating the time occurred outside of an interrupt, it would be natural to update the time-based counters at the same place; since it made sense outside the interrupt, I left the two together.
3) Updating portions of the display
   These portions of the display rely on the time and the do not need to be updated every time the display refresh is called. Overall we save operations.
4) Reason for all of this:
   We only call this function every 60,000 cycles, it does not take too many cycles, which does not effect the rotary encoder, the next interrupt that needs to be completed before it's time roles around again. There might be an issue with the sound that has not been addressed since I did not complete that portion.

Display

Buffer and refreshment

I keep a buffer, Display_Buf, which holds hold the values of each of the 7 LED display components: 4 seven segments, 1 for the punctuation (colon, apostrophe), and the one for the red and one for the green components of the LED's. The eighth one would have one bit for the sound output depending on the jumper.

The display needs to be refreshed fairly often, and is done so in the main loop (page 22) via a function called Refresh_LED (page $$). Refresh_LED simply takes what is in each buffer location and prints it to the actual display through the SPP port (page 12), but we only update one part of the display each time we get our call. We take care of ghosting by turning off the cathodes.

Issues

There is an issue of ghosting that I believe comes from a portion of my code that I have used to test a function that I have not removed, and have not found.

Writing to the buffer

Using a function called LED_Write (page $$) we can write to the display buffer with some organization. The two arguments are WREG and FSR0. WREG holds the value to be displayed and FSR0 holds both where to display that value and how to interpret the value in WREG. There are two ways to translate the WREG value:

1) Pre-coded for the display, using constants, we can submit as an argument the values that directly translate to a character on the display.
2) Translate the value in WREG as a JSCII (page 57), most useful for displaying a number, the first 16 values in the JSCII table are the numbers 0 through A, respectfully order.

The high byte of FSR0 controls this decision. The lower byte of FSR0 controls which part of the display our value goes. Its bit values correspond to each of the seven areas (Seven Segments, Green, etc).

The display buffer itself can be accessed, but should do write to buffer via LED_Write.

Constants (#DEFINE's)

We have several groups of definitions

- Hardware reference: cathode and anode addressed
- Anodes: which segments (A through G, decimal point, apostrophe, colon, matrix of LED's) will light when a zero is written to them
- Cathodes: which area (one of four seven segments, green LED's, etc) will light when a zero is written to them.
- Translate character values (JSCII) 0-9, A-Z, some characters, these are logical combinations of which segments will light up.

Switches

There are 12 switches on this board, 8 arrow buttons corresponding to 4 seven segment displays and 4 mode buttons corresponding to their name in Figure 2. We can identify up to 2 switches being pressed at the same time. If three or more are pressed, we do not get any more info, and only 2 will be reported as being pressed (the leftmost two, I believe).

Updating Values

Sw_State_16 (page $$) and Sw_State_7C (page $$) are updated to report which switches are currently being pressed at the time UpdateSwitches (page $$) is called. UpdateSwitches is called on a low priority interrupt (page 9) at a slow enough rate to avoid bouncing. Sw_PState_16/7C holds the previous state of which switches were pressed. This is used to see once a switch has been released, that it was in fact previously held down.

The states can then read Sw_State_16/7C and see if those switches are being pressed, if a function reads a switch, it should clear that corresponding flag (this gives the interrupt control of the 'refresh rate,' not whatever frequency the function in question is being called at).

QuerySwitchesII

While testing for the switches, I found that in each set of 4, every combination of pressing produced a unique response, see the Table 1 (without masking the response). (Switches 1-4 correspond to 5-8 and 9-C on the other blocks, respectively).

Table 1

Switch(es) Pressed	Value Read (Hex)
None	F
1	E
2	D
3	B
4	7
1 & 2	C
1 & 3	A
1 & 4	6
2 & 3	9
2 & 4	5
3 & 4	3
1 & 2 & 3	8
1 & 2 & 4	4
1 & 3 & 4	2
2 & 3 & 4	1
All	0

This means that each switch corresponds to a bit in the result (what is read from SPPDATA).

A '1' is read when the switch is not pressed, a 0 0 0 0 is read
↑ ↑ ↑ ↑
when switch 4 3 2 1 is pressed.

My clock design will not have an action attributed to having two switches pressed in the same 'block' at the same time, so I will look only at two switches being pressed in different blocks (except for rightmost block, there two can be pressed).

Identifying how switches have been pressed (in states/functions outside of the interrupt)

The procedure for identifying if a switch has been pressed and released and/or held down:

1) Determine if the switch is currently being held down (test a bit in Sw_State_16 for switches 1-6 or in Sw_State_7C for switches 7 – C).
   a) If it is being pressed:
      i) There is a counter, HDCount, that keeps track of how many hundredths of a second have passed. This should be set once a switch has been pressed, we do so here if needed.
      ii) See if that counter has reached zero, if so, then that switch has been held down for the proper time to classify as being "Held Down."
         (1) If HDCount has reached zero:
            (a) Disable counter.
            (b) Execute code that should happen after the switch has been held down.
         (2) Else exit
   b) Else it is not being pressed, we need to see if had been pressed via Sw_PState_16/7C, to see if it had been held down previously (every time Sw_State_xx gets updated, the old value gets stored in Sw_PSate_xx).
      i) If it had not previously been pressed, then another switch may have been previously pressed, exit.
      ii) If HDCount is not set, then no switch has been pressed, exit.
      iii) Else: we had been pressing our switch, which means HDCount has started, and we need to 'clear' it. We also do whatever needs to be done when this switch has been 'Pressed and Released.'

There is an example on the following page.

Example switch code, this example will be using switch 9, switch 9 corresponds to button 't' (time):

```
 BTFSS Sw_State_7C, 9 - 6, 0         ;; see if switch 9 pressed
 BRA   State00_release_t             ;      skip if not pressed
 BTFSC HDCount, 7, 0                 ;      Set HDCount if it is
 CALL  SetHDCount                    ;       negative (not running)
 TSTFSZ       HDCount                ;      if zero, Held down t
 BRA   State00_t_undecided           ;;     ELSE go on with other stuff

 ;; if here t is being held down
 SETF  HDCount, 0                    ;; disable counter

 ;; Do what you want when t is being held down.

 BRA   State00_t_undecided

State00_release_t:
 ;; if here, see if t was pressed AND not counting HD
 ;;    if so: to goto state 01
 BTFSS Sw_PState_7C, 9 - 6, 0        ;; see if switch 9 was pressed
 BRA   State00_t_undecided           ;; if not ever pressed
 BTFSC HDCount, 7, 0                 ;; see if HDCnt disabled
 BRA   State00_t_undecided           ;; if not ever pressed

 ;; if here, then we had pressed 't' and released it before
 ;;    it got to the 'held down' state.
 SETF  HDCount, 0                    ;; since let go, disable counter

 ;; Do what you want when t has been pressed and released.

 ;; done looking at 't' right now.
State00_t_undecided:
```

End of example.

Rotary Encoder

The rotary encoder is also updated via interrupt (page 9). When RE_intrpt (page $$) is called, it looks at the rotary encoder info on the SPP port, compares it to the previous state, RE_state (page $$), and then updates RE_POS (page $$) accordingly.

Clockwise rotations corresponds to RE_POS being incremented, and conversely.

Changing States

As I have said before, states control the unique happenings within our alarm clock. That being said, there are two ways to change states, the first is to let a time run out, the second is to manually change states based on user input. Some states employ both of these transitions.

Changing based on the timer

One of the functions of the main loop is to see if it is time to change to a new state, this is done in the function, Time_Change_State (page $$). Time_Change_State looks at DelayCount (page $$) to see if it is time to change to a new state (when it is zero). If it is zero, we set our next state to the value kept in DelayState (page $$), and we disable the counter (make it negative), we also change the DelayState value to zero, the default state as an error precaution.

Setting up DelayCount and DelayState should be done in the **preceding** state. That is, if we want to display 'dAtE' for 2 seconds, the preceding state needs to set DelayCount to 0xC8 = 200 and DelayState to [current state] before the transition. All that the current state needs to do is update the display. The current state is unable to set these delay values (without much trickery), without trickery, we would consistently set the delay counter, and it would never reach zero. *The current states can also leave based on user input, but in doing so, they should clear the delay registers, otherwise when the time runs out, we will leave the current (next) state and go to the one specified by DelayState.*

Downside: we cannot have two passive states occur after each other. The way around this is to place a third state between the two passive ones. This third state will set up the delay registers for second passive state, and then set the current state to that second state

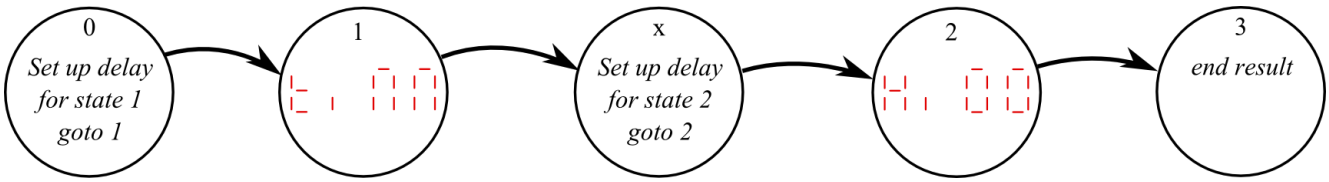


Figure 3

Changing based on user input

We have seen how to interpret the switch state registers (page 12) over time, we uses that code in each state. For the most part, this code is copied in each state, since it needs to be modified so much, perhaps a series of macros would be more appropriate, but the overall effect is the same.

There are three ways for a state to respond to a switch being pressed, the state can respond in two separate ways to the second to responses below.

1) Do nothing (do not even check its bit values).
2) Wait for it to be held down (held down counter expires)
3) See that it was pressed and released (released before held down counter expirers)

A couple of the states make use of the fact that a switch was held down to get to their state, and they just stay in their state as long as that switch is being held down (the set current time states do this).

Though most user input is state specific, a couple types of user input occur in multiple states.

Pressing 't' to escape to default mode (from just about ever state)

A function call to QreturnToState00Q (page $$) will check to see if the switch corresponding to the 't' button is pressed (does not care if released, only pressed). If so, the next state will be state30 (page $$) and the delay state is set to state00 (page $$), the default state. We go to state30, so we can see the display of the default state, but we do not register any button presses, since 't' will still be held down. (it is noticed almost instantly, and microseconds later, we are in a different state, before a human can physically release the button). We are only in the phony state for a couple hundredths of a second, but it is enough. The user would have to act pretty quickly to want to press other buttons, believing they are in the default state and find them not working.

Holding down 'a' to change to next alarm in list (from states that view the alarm stack (page 27))

A call to QchangeCurrentAlarmQ (page $$) will look to see if the 'a' button is being held down, if it is held down long enough, we update the alarm pointers to view the next alarm in the list, and are sent to the opening state of the alarm review sequence. If we are adding an alarm, and have not seen 'SET' then we through away the current un-added data.

Holding down 'd' to delete the current alarm (from state that view the alarm stack (page 27))

Similar to the previous, except we are pointed to states that prompt the user to confirm that they want to delete their alarm. The function called is QdeleteAlarmQ.

Responding to the yes or no question

When YesOrNo (page $$) is called, it returns a value saying whether the user has pressed the switches under the y or the n on the screen (or has no pressed either buttons). This function is different from the ones above it, in that it also manages the display portion. States that call this should not attempt to display (update the display buffer) themselves, which ever is done second will be what is actually displayed, but this wastes time.

Using the \/\ to change the current values

This is used to change the values what is currently being displayed, i.e. while viewing minutes and seconds we press the leftmost up arrow, we will increment the tenth's place of the minutes. This is done by the complex function MT_MD_Change_q (page $$) which can change minutes, hours, seconds, and the date; there is also one that behaves similarly, but only changes the current day of the week, MD_Change_q (page $$).

MT_MD_Change_q takes a code as an argument, and based on that code responds differently when certain switches are pressed. This list of codes in this function's description (page $$).

Alarms

There can be up to 32 alarms in the processor stored on a stack, starting at Alarm_Base_addr. Each alarm can ring once or every day (initially we wanted to be able to ring certain days of the week, but this was never accomplished, but still achievable). Alarm can be set to the minute, and will ring when seconds go from 59 to 00. Alarms ring only when the date they are set to matches the current date (and the time matching too). This means when setting an alarm, you set the date to the first time you want the alarm to ring. But as you will see below, we can only store one date per alarm, so how to have that alarm repeat?

There is also a modification stack for the alarms, starting at Alarm_Mod_Base_addr with entries corresponding to entries in the alarm stack. When an alarm rings, we look at its corresponding modification data to see if we need to repeat or not, if we need to repeat, we alter the alarm's date so that it rings the next time (as of now, this is updating the date to ring the next time). This will be described a little better below.

Exceptional Alarms

We have two exceptional alarms: the daily alarm and the snooze/nap alarm. These are located at the base of the stack, and each is modified outside of the normal alarm modification routine, in fact, they are not even viewable with the rest of the alarms (page 25).

The daily alarm's modification data says that it will ring the next day, but can be turned off through a separate routine (state33 on page $$) and viewed and altered easily (states 03, 13 pages $$, $$).

The snooze alarm is never seen by the user, and is set based on the current time plus the default snooze time when the snooze button is hit (while alarm is ringing), or we enter nap mode (page $$). It is not modified after it rings.

All the other alarms are called 'special alarms'

Data Structure of an Alarm

4 bytes, each in compact BCD form.

date [high]	date [low]	hours	minutes

Structure of an Alarm Modification

4 bytes in varying forms.

modification value [high]	modification value [low]	*used as temporary storage*	modification code

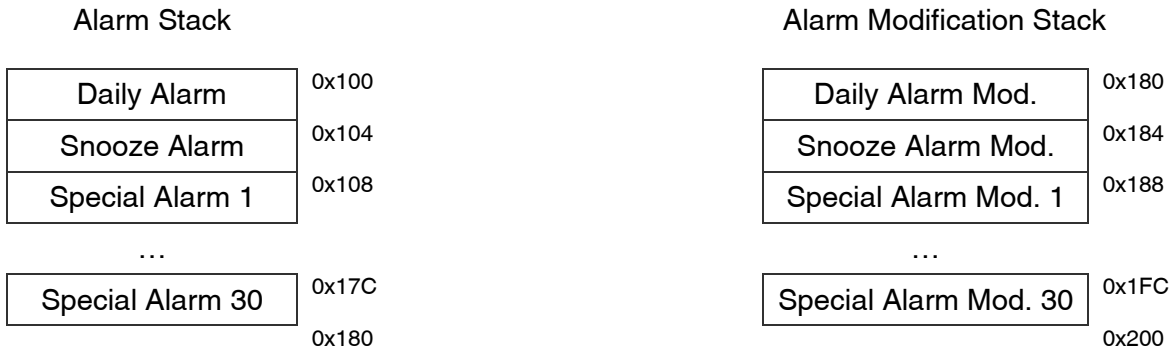
The modification codes determine what the higher two bytes mean:

Modification Code	Result
0x00	Don't modify
0x01	Add larger two bytes to the alarm's date (will only be 1 for this program)
0x02	Use larger to bytes as flags for which day of the week to ring next, look at MD to calculate. (not implemented)
0xFF	Alarms has not been written yet.

As you can see, other codes are available for implementation. Right now we just use 0x00, 0x01, and 0xFF.

Alarm and Modification Stack

Using the constant definitions I actually implemented:



Alarm Menu

Pressing 'a' from the default mode will take you to the alarm menu, which gives the user access to the special alarms by adding one or reviewing the list and default nap and snooze lengths. The alarm menu (states on pages 26, 27) is a series of passive states that display or prompt user input that takes place in the following active state. These are really best seen in their state diagrams and explanations, also see the User's Manual for an external view of how they work.

Initialization

After the boot menu occurs, we are tasked with initializing registers and interrupts as well as resetting different systems. 'Each' system has a reset function to call to initialize its respective registers.

Register Initialization

We must set the current state, MASTER_STATE (page $$), to the default, zero. This also turns off the alarm ringing and makes us not snoozing (see Super States on page 25).

The delay state counter, DelayCount (page $$), needs to be turned off (which means negative).

The delay state register, DelayState (page $$) itself is set to the default state for safety.

The switch counter that indicates if a switch has been held down long enough is turned off (page 15).

Reset Calls

Reset_LED (page $$) sets up the values that allow the display refresh routine to rotate through each display area. This also prepares the boot up display (Displays "JOHN").

LowIntReset (page $$) allows us to use the interrupt dispatcher.

Reset_MT sets up the counter to the interrupt to overflow every 60,000 instructions (page 11) and initial time/date are stored (these are arbitrary, but have some rules, see the Reset_MT code).

resetSwitches (page $$) prepares the switches to be updated on their interrupts

RE_reset (page $$) initializes the rotary encoder state

AlarmReset (page $$) sets up the alarm stack and modification stack pointers (page $$) as well as setting the modifications of the daily and snooze alarm.

Interrupts

These need to be set after the reset calls are made, since the interrupt routines rely on initialized registers. These interrupts use the TmrIntrpt_setup macro to initialize themselves, see the code (ENEE 440 – PIC18 Alarm Clock Main.asm) for a in depth explanation.

Timer 0 interrupt (low) is for the switches. Need a low frequency, so we use the 16 bit counter and set the prescale to 2^3, which yields ~22.9 Hz. Quick enough so the user cannot press the button too quickly (they would have to hold it down for less that 40 ms, and on the correct interval) but slow enough to avoid bouncing.

Timer 1 interrupt (high) needs to be slow as well, we choose to its prescale to be 2 (remember the interrupt code itself alters the counter to get 0.01 second period).

Timer 2 interrupt (low) is for the rotary encoder, we scale it a bit (pre and post) to run quicker then a person can turn the dial, since we don't have code to correct for both bit changing between readings.

Timer 3 interrupt (low) was for the sound, but this was never finished.

Main Loop

I called my main loop IdentidemVicis (Latin). As seen on page 8, the main loop's tasks are to refresh the display, react to special alarm circumstances, and pick which state code we will execute. Some code gets executed every time (refreshing the LED's and checking to see if the alarm is ringing) everything else happens within separate states, so the main loop needs to decide which state code to run (including super states (page 25)).

Refreshing the Display

This has already been discussed in the display section on page 12. I tried to do this in an existing interrupt, but it did not refresh correctly, but this was a quick trial, I may have done it incorrectly. It stays in the main loop for now.

Managing Alarms

The code in Manage_Alarms (page $$) checks to see if an alarm is ringing for the first time, if so, we set a bit in MASTER_STATE (page $$) that indicates the alarm is ringing, the code that reports a new alarm ringing also modifies the ringing alarm so that it will ring (or not ring) the next day. If an alarm is ringing we set the current state to the default state because we want to see the current time.

We also want to respond to the user, so they can turn off the alarm or enter snooze. We use the lower byte of DelayState to store rotary encoder position when the alarm rings. If that value changes, we enter the snooze mode. We also look for the 'a' or 's' button to turn off the alarm or enter snooze mode (respectfully). Turning off the alarm is as simple as deasserting the alarm ringing bit of MASTER_STATE and going to state30 (page $$) (so we do not register 'a' as being pressed in the default state). Entering snooze mode involves calculating the snooze alarm time and date by adding the current time and date with the default snooze time and storing this information in the alarm stack; we also need to assert the snoozing bit in MASTER_STATE.

Managing in Snooze Mode

If the snooze bit of MASTER_STATE (page $$) is set, we do not want to execute any of the states' code. So we check that bit and skip the snooze code if we are not snoozing. Otherwise we execute Manage_Snooze (page $$). Entering nap mode also has this effect, but we have set the snooze alarm calculating in the default nap time in place of the default snooze time, see state04 (page $$).

Manage_Snooze displays the time left of the snoozing session. This is calculated by subtracting the time/date of the alarm and the current time. We must also respond to the rotary encoder. Turning clockwise means adding more time to the alarm (and conversely). After adjusting the time, we need to see if the alarm time agrees with the current time, if so, the user has dialed to zero minutes left. This allows us to exit the snooze mode, without the alarm ringing (since we will not be interrupted to check to see if the alarm is ringing, it is not possible for it to accidentally ring unless the time actually expires with 1 minute left as the user is about to dial down, then we don't actually have an error, the user might be annoyed though, in fact the user will probably enter snooze mode right away. There is not a great way to check for this, since we cannot predict what the user will do).

Switched States Based on Delay Timer

As talked about on page 17, if DelayCount (page $$), both high and low bytes, are zero, then we need to switch states to the one recorded in DelayState (page $$) as well as turning off the DelayCounter.

Going to a State's Code

SeekState (page $$) contains code to take the value in MASTER_STATE and use it as an offset to the current program counter to jump to a GOTO command. I used a jump tabled indexed by the MASTER_STATE value. Calculating the offset PC involves pushing the current PC onto the stack, so we can edit it. We then can shift MASTER_STATE's value, and add in the number of instructions that are between the push and the beginning of the actual code. After this is done, we just need to update the pushed PC value to the current PC value (RETURN instruction).

States

Figure 4, Figure 5 and Figure 6 contain state diagrams that completely outline the special workings of this processor. All of the code pertaining to the states are in
"ENEE 440 – PIC08 Alarm Clock Main.asm"

Explanation

States are stored in MASTER_STATE (page $$). The higher two bits are set if the alarm is currently ringing or if we are in snooze mode (respectfully). When either is high we execute super state code, code that will happen despite which actual state we are in. When snoozing, we do not use any of the normal states. When the alarm is ringing, along with executing special code, we also execute part of the code in state00 (page $$) so that we may see the current time, but cannot leave this state.

The lower 6 bits are the normal state numbers (seen in the top center of the state bubbles). The super states are in Figure 4 as well as how to read the state diagram figures.

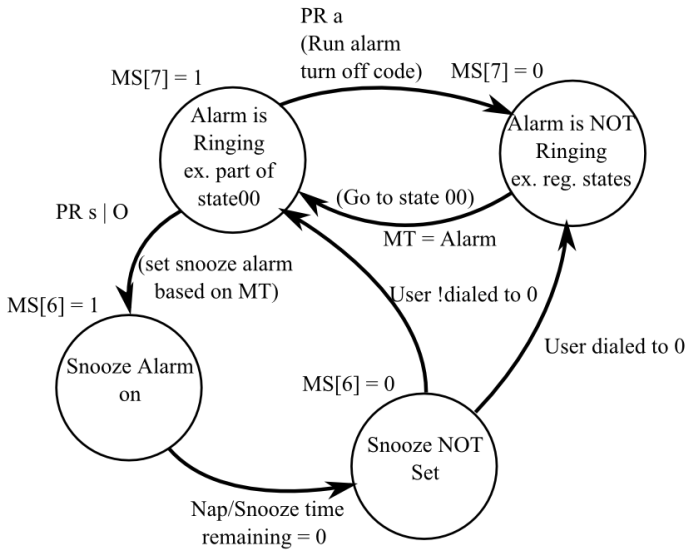
State List Format

State##	
Comes from	Summary of which states can get to this one
Goes to	Summery of which states can be arrived at from this one
Display	What is on the LED's. If not stated, the colon ':' is off. If not states, the green and red LED's are either off or are carry their previous value from the previous state
Description	What this state does. Idle states use the delay state system to get out of their state.
Register Notes	Which registers are or are not effected by this state (when exception to their general use occurs)

State Diagram Page 0

Key:
- PR - Push and release
- HD - Hold down
- W - wait for a certain time to pass
- O - rotary dialed
- t - time button (escape)
- d - daily alarm button
- a - alarm (others) buttons
- s - set button
- \/\ - corresponding up/down button
- & - AND
- | - OR

Super States



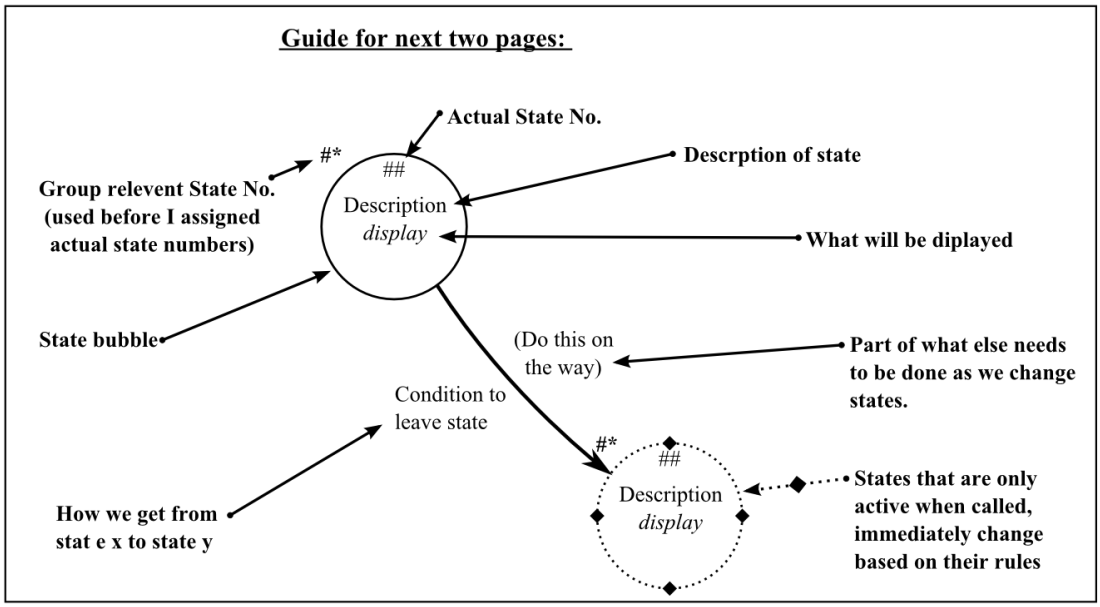


Figure 4

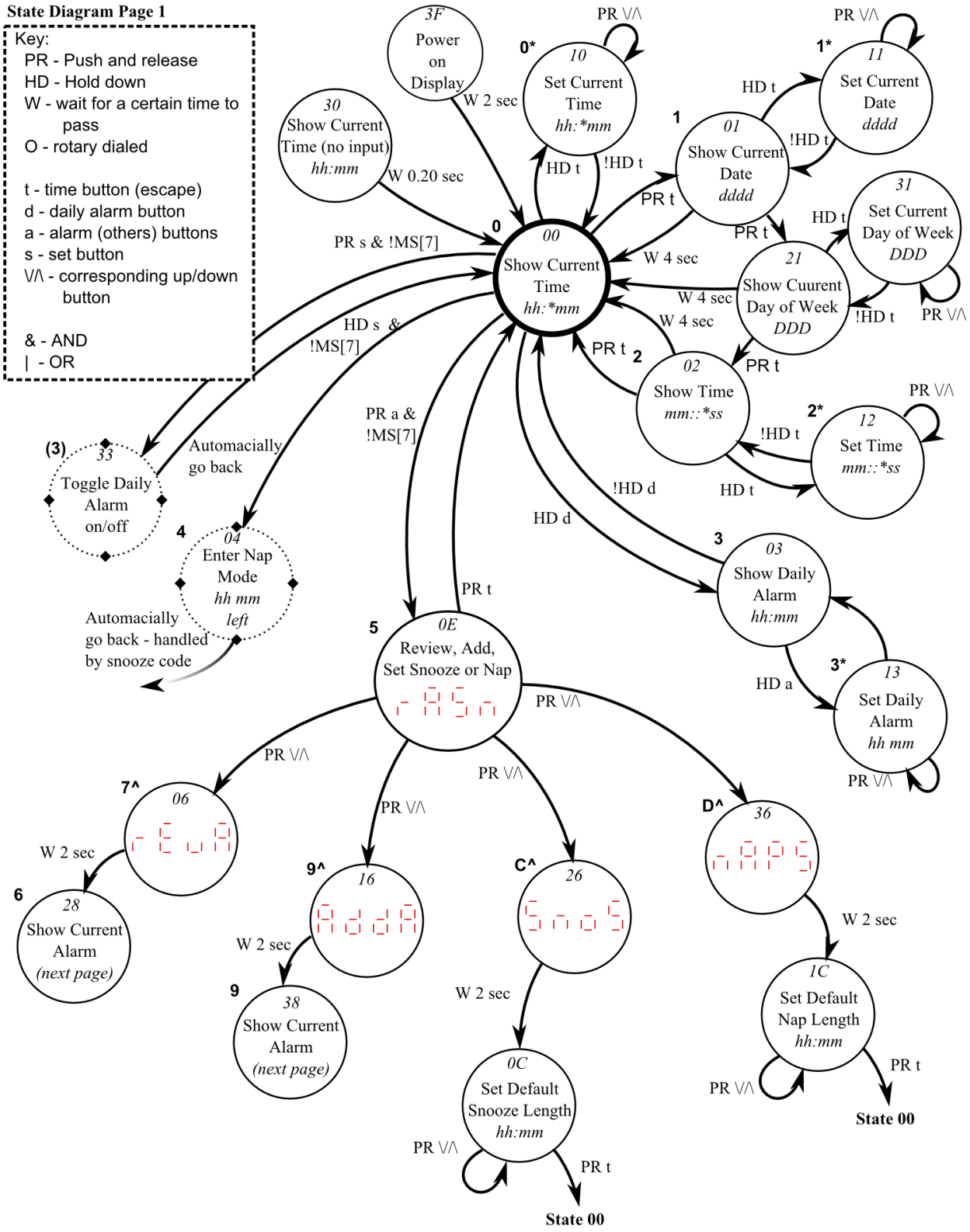

State Diagram Page 1
Key:
PR - Push and release
HD - Hold down
W - wait for a certain time to pass
O - rotary dialed
t - time button (escape)
d - daily alarm button
a - alarm (others) buttons
s - set button
\/\ - corresponding up/down button
& - AND
| - OR
3F Power on Display
W 2 sec
30 Show Current Time (no input) hh:mm
W 0.20 sec
0* 10 Set Current Time hh:*mm
PR \/\
HD t
!HD t
1 01 Show Current Date dddd
1* 11 Set Current Date dddd
!HD t
HD t
PR t
31 Set Current Day of Week DDD
!HD t
PR \/\
0 00 Show Current Time hh:*mm
PR s & !MS[7]
HD s & !MS[7]
W 4 sec
21 Show Cuurent Day of Week DDD
W 4 sec
PR t
2 02 Show Time mm::*ss
2* 12 Set Time mm::*ss
!HD t
HD t
PR \/\
(3) 33 Toggle Daily Alarm on/off
Automacially go back
4 04 Enter Nap Mode hh mm left
Automacially go back - handled by snooze code
PR a & !MS[7]
!HD d
HD d
3 03 Show Daily Alarm hh:mm
3* 13 Set Daily Alarm hh mm
HD a
PR \/\
5 0E Review, Add, Set Snooze or Nap
PR t
PR \/\
7^ 06
W 2 sec
6 28 Show Current Alarm (next page)
9^ 16
W 2 sec
9 38 Show Current Alarm (next page)
C^ 26
W 2 sec
0C Set Default Snooze Length hh:mm
PR \/\
PR t
State 00
D^ 36
W 2 sec
1C Set Default Nap Length hh:mm
PR \/\
PR t
State 00

Figure 5

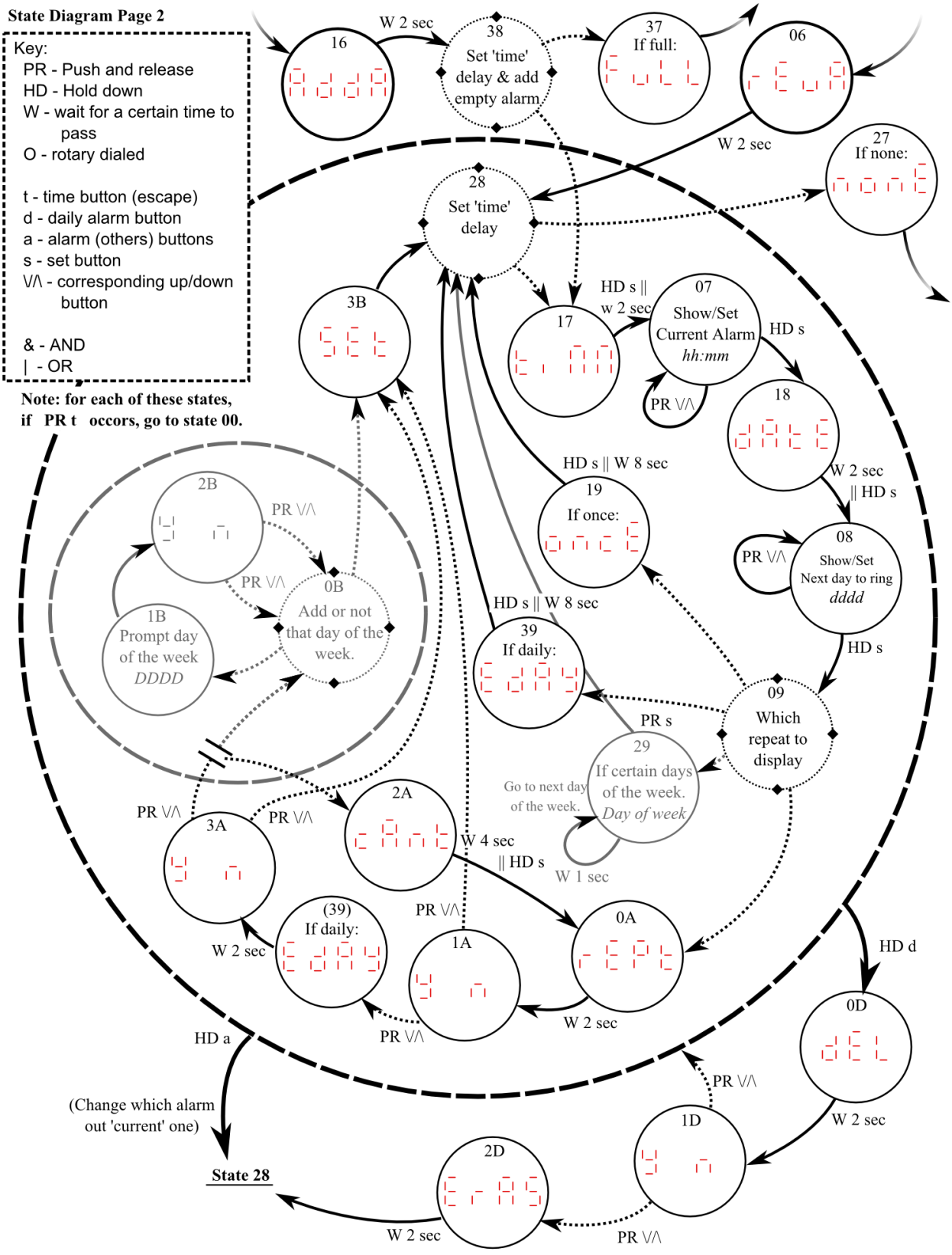

State Diagram Page 2
Key:
PR - Push and release
HD - Hold down
W - wait for a certain time to pass
O - rotary dialed
t - time button (escape)
d - daily alarm button
a - alarm (others) buttons
s - set button
\/\ - corresponding up/down button
& - AND
| - OR
Note: for each of these states, if PR t occors, go to state 00.
16
W 2 sec
38
Set 'time' delay & add empty alarm
37
If full:
06
W 2 sec
27
If none:
28
Set 'time' delay
3B
17
HD s || w 2 sec
07
Show/Set Current Alarm hh:mm
HD s
PR \/\
18
W 2 sec || HD s
HD s || W 8 sec
19
If once:
08
Show/Set Next day to ring dddd
PR \/\
HD s
HD s || W 8 sec
39
If daily:
09
Which repeat to display
PR s
29
If certain days of the week. Day of week
Go to next day of the week.
W 1 sec
2B
PR \/\
0B
Add or not that day of the week.
1B
Prompt day of the week DDDD
2A
W 4 sec || HD s
3A
PR \/\
(39)
If daily:
W 2 sec
1A
PR \/\
0A
W 2 sec
HD d
0D
PR \/\
W 2 sec
1D
PR \/\
2D
W 2 sec
HD a
(Change which alarm out 'current' one)
State 28

Figure 6

State3F – Opening Message	
Comes from	Reset
Goes to	Default state (State00)
Display	Opening message
Description	Idles until delay count expires

State30 – View Current Time (hh:mm)	
Comes from	escaping out of state where we don't want a pressed button to execute in the default state (State00)
Goes to	Default state (State00)
Display	Current time's hours and minutes; flash : at 1 Hz 'CURRENT TIME' LED is red
Description	idles until display count expires

State00 – Default State	
Comes from	Many places
Goes to	Setting the current time, toggle daily alarm, view daily alarm, view current date, enter alarm menu, enter nap mode. States 10, 33, 03, 01, 0E, 04
Display	Current time's hours and minutes; flash : at 1 Hz 'CURRENT TIME' LED is red
Description	Default state, watches four mode buttons for being both pressed and released and held down. Goes to different states depending on input. This is where get when you escape from another state, and how you start your path to any other state. When the alarm is ringing, only the display part of the code is executed.
Register Notes	Doesn't effect HDCount while alarm is ringing

State10 – Set Current Time (hh:mm)	
Comes from	Viewing the current time (State00) by holding down 't'
Goes to	Viewing the current time (State00) by releasing 't'
Display	Current time's hours and minutes; flash : at 1 Hz 'CURRENT TIME' LED is red; 'SET TIME' LED is red
Description	When holding down 't', pressing \/\ will change the current time's hours and minutes

State01 – View Current Date	
Comes from	Viewing current time (State00) by pressing 't'
Goes to	setting current date, viewing current day of week, default state (States 11, 21, 00) by holding down 't', pressing 't' or waiting 8 seconds (respectfully)
Display	Current Date; 'CURRENT TIME' LED is red
Description	Displays the current date

State11 – Set Current Date	
Comes from	Viewing current date (State01) by holding down 't'
Goes to	Viewing the current date (State00) by releasing 't'
Display	Current date 'CURRENT TIME' LED is red; 'SET TIME' LED is red
Description	When holding down 't', pressing \/\ will change the current date

State21 – View Current Day of the Week	
Comes from	Viewing current date (State01) by pressing 't'
Goes to	setting current day of the week, viewing current time in minutes and seconds, default state (States 02, 31, 00) by holding down 't', pressing 't' or waiting 8 seconds (respct)
Display	Current day of the week; 'CURRENT TIME' LED is red
Description	Displays the current day of the week

State31 – Set Current Day of the Week	
Comes from	Viewing current day of the week by holding down 't'
Goes to	Viewing the current day of the week by releasing 't'
Display	Current day of the week 'CURRENT TIME' LED is red; 'SET TIME' LED is red
Description	When holding down 't', pressing \/\ will change the current day of the week

State02 – View Current Time (mm:ss)	
Comes from	Viewing current day of the week (State21) by pressing 't'
Goes to	setting current day of the time in minutes and seconds, viewing current time in hours and minutes, default state (States 12, 00, 00) by holding down 't', pressing 't' or waiting 8 seconds (respectfully)
Display	Current Time's Minutes and Seconds 'CURRENT TIME' LED is red
Description	Displays the current time's minutes and seconds

State12 – Set Current Time (mm:ss)	
Comes from	Viewing current time's minutes and seconds (State02) by holding down 't'
Goes to	Viewing the current time's minutes and seconds by releasing 't'
Display	Current Time's Minutes and Seconds; flash : at 5 Hz 'CURRENT TIME' LED is red; 'SET TIME' LED is red
Description	When holding down 't', pressing \/\ will change the current time's minutes and seconds

State33 – Toggle Daily Alarm On/Off	
Comes from	Default state (State00) by pressing and releasing 's'
Goes to	Delay to default state (State30)
Display	-
Description	Toggles the daily alarm on/off. Turns on by changing its date to the current date if alarm's time > current time or changing its date to the next day of alarm's time < current time. To turn it off, just put alarm's date to 0000. *Only here for one cycle*

State04 – Initiate Nap (Snooze) Mode	
Comes from	Default state (State00) by holding down 's'
Goes to	While snooze (nap) is active, we don't go to states
Display	This state doesn't display anything, the snooze super state will display the time left in the snooze session
Description	Starts snooze session by turns on the snooze mode as well as setting the snooze/nap alarm to current time/date + default nap time.

State03 – View Daily Alarm	
Comes from	Default state (State00) by pressing 'd'
Goes to	setting daily alarm's hours and minutes or the default state (States 13, 00) by holding down 'd', pressing 't' or waiting 8 seconds
Display	Daily alarm's hours and minutes; flash : at 1 Hz 'REVIEW/EDIT' LED is red
Description	View the daily alarm's time

State13 – Set Daily Alarm	
Comes from	Viewing daily alarm's time (State03) by holding down 'd'
Goes to	Viewing the daily alarm's time by releasing 't'
Display	Daily Alarm's time: hours and minutes; flash : at 1 Hz 'REVIEW/EDIT' LED is red; 'SET ALARM' LED is red
Description	When holding down 'd', pressing \/\ will change the daily alarm's time

State0E – Alarm Menu	
Comes from	Default state (State00)
Goes to	Review Alarms, Add Alarms, Snooze set, Nap set, or default state (states 06, 16, 26, 36, 00) by pressing the corresponding /\\/ or 't'
Display	"r A S n" = (see above cell)
Description	Presents a menu to enter alarm menu system, make menu choices by pressing the arrow underneath whichever 7seg matches.

State06 – Display 'Review'	
Comes from	Alarm menu (State0E)
Goes to	Set 'time' delay (State28)
Display	"ReVA" = review alarm
Description	Idle while displaying message

State28 – Prepare to View Special Alarms	
Comes from	Review Alarm message, alarm set confirmation, or viewing alarms frequency (states 06, 3B, 19/39) by waiting or holding down 's' Or by any o the view/set special alarm states by holding down 'a'
Goes to	If there are no special alarms: display "none" (State27) Else: display "time" then display alarm's time (State17)
Display	-
Description	Figure out if there are special alarms in the stack, if so, continue to display the time of the current special alarm *Only here for one cycle*
Register Notes	Edits FSR1 and FSR2 within rules specified

State27 – Display 'None'	
Comes from	State28
Goes to	Default state (State00)
Display	"nonE"
Description	Idle while displaying message

State16 – Display 'Add Alarm'	
Comes from	Alarm menu (State0E)
Goes to	Set 'time' delay and add empty alarm (State38)
Display	"AddA" = add alarm
Description	Idle while displaying message

State38 – Prepare to Add a Special Alarm	
Comes from	Review Alarm message by waiting or holding down 's'
Goes to	If there are 30 special alarms: display "full" (State37) Else: display "time" then display alarm's time (State17)
Display	-
Description	Figure out if there are special alarms in the stack, if so, continue to display the time of the current special alarm *Only here for one cycle*
Register Notes	Edits FSR1 and FSR2 within rules specified

State37 – Display 'Full'	
Comes from	State38
Goes to	Default state (State00)
Display	"FULL"
Description	Idle while displaying message

State17 – Display 'Time'	
Comes from	Set time delay with/without adding an empty alarm
Goes to	Showing/setting current alarm's time (State07)
Display	"tiM" = time 'REVIEW/EDIT' LED is green
Description	Idle while displaying message

State07 – Edit Current Alarm's Time	
Comes from	Time display (State17) by waiting or holding down 's'
Goes to	Displaying date (State18) by holding down 's'
Display	Current alarm's time (hours and minutes) : flashes at 1 Hz 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	Change the current alarm's time by pressing and releasing /\/
Register Notes	Edits FSR1 and FSR2 within rules specified

State 18 – Display 'Date'	
Comes from	Viewing/Setting Current Alarm's time (State07)
Goes to	Viewing/Setting Current Alarm's date (State08)
Display	"dAtE" = date 'REVIEW/EDIT' LED is green
Description	Idle while displaying message

State08 – Edit Current Alarm's Date	
Comes from	Date display (State18) by waiting or holding down 's'
Goes to	Which frequency state (State09) by holding down 's'
Display	Current alarm's date 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	Change the current alarm's date by pressing and releasing /\/. This is when the next time the alarm will

	ring (can not ring for several days, even if alarm is set to ring every day, once it rings, it will ring every day)
Register Notes	Edits FSR1 and FSR2 within rules specified
State09 – Which Frequency State to Go To	
Comes from	Viewing Alarm's Date (State08) by holding down 's'
Goes to	States 19, 39 (29) or 0A depending
Display	-
Description	Look's at the current alarm's modification data in the modification stack, based on that value, go to: 0x00 once state19 0x01 everyday state39 0x02 certain days of the week (unimplemented) 0xFF not set yet state0A
Register Notes	Edits FSR1 and FSR2 within rules specified

State19 – Display 'Once'	
Comes from	State09
Goes to	State28 (Start cycle over again)
Display	"oncE" = once 'REVIEW/EDIT' LED is green
Description	Idle while displaying message

State39 – Display/Prompt 'Every Day'	
Comes from	State09
Goes to	State28 (Start cycle over again)
Display	"EdaY" = Every Day/Every Day? 'REVIEW/EDIT' LED is green / 'SET ALARM' LED green
Description	Idle while displaying message

State0A – Prompt 'Repeat'	
Comes from	State09
Goes to	Yes or No (State1A)
Display	"rEPt" = Repeat? 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	Idle while displaying message

State1A – Yes or No (repeat)	
Comes from	Repeat? (State0A)
Goes to	Display Set (State3B) or Prompt Every Day (State39)
Display	"Y n " = Yes or No 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	If user pushed \/\ for yes, continue to state 39, then 3A If user pushed \/\ for no, add 4 to stack pointers (FSR1, FSR2), effectively adding this alarm to the stack
Register Notes	Edits FSR1 and FSR2 within rules specified

State3A – Yes or No (Every Day)	
Comes from	Every Day? (State39)
Goes to	Display Set (State3B) or Display Can't (State2A)
Display	"Y n " = Yes or No 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	If user pushed \/\ for no, continue to state 2A, since we cannot have an alarm that both repeats AND does not ring every day, originally we would continue to state 0B where we would start the process of prompting which days of the week we want this alarm to ring. If user pushed \/\ for yes, add 4 to stack pointers (FSR1, FSR2), effectively adding this alarm to the stack
Register Notes	Edits FSR1 and FSR2 within rules specified

State2A – Display 'Can't'	
Comes from	State3A
Goes to	Prompt Repeat (State0A)
Display	"cAnt" = can't 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	Idle while displaying message, we then ask about repeating again until the user repeats every day or not at all (or exits before this alarm is saved)

State3B – Display 'Set'	
Comes from	No/Yes (State 1A, 3A) respectfully (0B if implemented)
Goes to	Start display of alarms beginning (State28)
Display	"SEt " = set; 'REVIEW/EDIT' LED is green
Description	Idle while displaying message

State0D – Prompt 'Delete'	
Comes from	Any of the special alarm viewing/setting states by holding down 'd'
Goes to	Yes or No (State1D) 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Display	"DEL " = Delete?
Description	Idle while displaying message

State1D – Yes or No (Delete)	
Comes from	Prompt 'Delete'
Goes to	Beginning of viewing/setting special alarms or back to the state which existed previous to arriving at State0D
Display	"Y n " = yes or no 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	If user says yes, then delete the current alarm by moving everything else in the stack down one alarm space and decrementing the alarm points by 4, then display 'erased' If user says no, go back to previous state (held in temporary portion of alarm's modification entry, the 2nd lowest byte)
Register Notes	Edits FSR1 and FSR2 within rules specified

State2D – Display 'Erased'	
Comes from	Yes or No (State1D)
Goes to	Beginning of reviewing alarms (State28)
Display	"ErAS" = Erased 'REVIEW/EDIT' LED is green; 'SET ALARM' LED green
Description	Idle while displaying message

State26 – Display 'Snooze Set'	
Comes from	Alarm Menu (State0E)
Goes to	Set default snooze length (State0C)
Display	"SnoS" = Snooze Set (default)
Description	Idle while displaying message

State0C – Set Default Snooze Length	
Comes from	Display 'snooze set' by waiting
Goes to	Default state by pressing 't' or 's'
Display	Default snooze length, hours and minutes
Description	Use \/\ to adjust default snooze length, automatically saved

State36 – Display "Nap Set"	
Comes from	Alarm Menu (State0E)
Goes to	Set default nap length (State1C)
Display	"nAPS" = Nap Set (default)
Description	Idle while displaying message

State1C State0C – Set Default Nap/Timer Length	
Comes from	Display 'nap set' by waiting
Goes to	Default state by pressing 't' or 's'
Display	Default nap/timer length, hours and minutes
Description	Use \/\ to adjust default nap length, automatically saved

Hardware

Input and Output

The input and output of this device used the SPP ports of the PIC. On the other side of the PIC there are eight external registers that hold the output values. Writing an address (writing to SPPEPS) will pick which register we want to read or write from. Reading from the SPPDATA for the switches/rotary encoder is not filtered through a register, but we can think of a register external that we get our data from.

We can see how to write an address, write data, and then read data (from Microchip) in Figure 7.

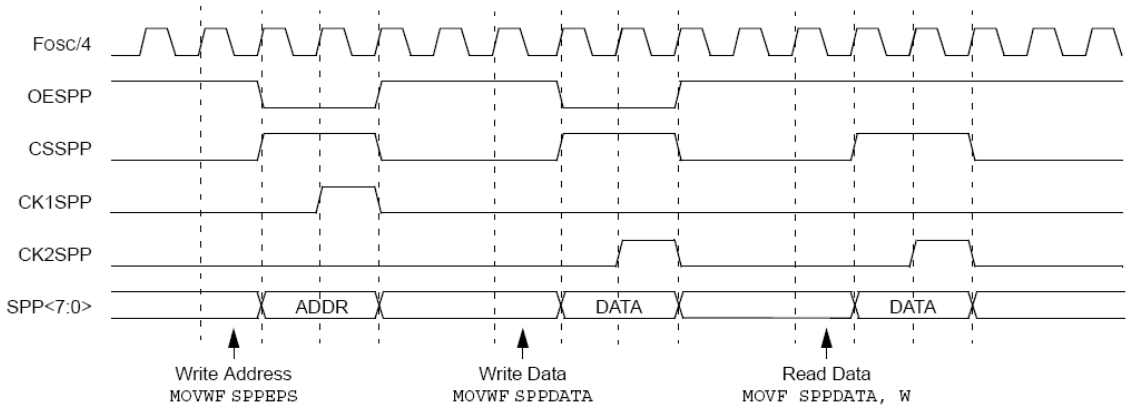


Figure 7

When reading and writing data, we need to confirm that our transmission is complete. This is done by checking the fourth bit of SPPEPS until it is clear. Writing addresses, data and reading data is done in two macros: MOVWSPP, MOVSPPW (use MOVWSPP for both address and data) see pages $$, $$.

Here are the steps to writing to the 7seg (set both anodes and cathodes):

```
MOVLW       [code to which area to write to]    ;; all 1's except area we want lit
MOVWSPP     ANODES (a literal)                  ;; 0x02
MOVLW       [which lights are on/off, a letter/number] ;; 1 = off, 0 = on
MOVWSPP     CATHODES (a literal)                ;; 0x01
```

Here are the steps to read from the switches

```
MOVLW       [code to which switches to allow]   ;; 0 for allow
MOVWSPP     SWITCHES (a literal)                ;; 0x00
MOVSPPW     SWITCHES (a literal)                ;; W holds 0 for switches pressed
;; you will need to mask of other bits
```

Note, these do not address the issue of ghosting, the rotary encoder is similar, but you do not need to write to the address first (its address is the same as SWITHCES) see the code comments for why which bits are which.

Clock

Our processor is driven by a 40 MHz crystal, but is divided/multiplied to 48 MHz for the speed of the processor. Actual instructions take this down by a factor of four.

Non-Obvious Methods

This section is to address the not-so-apparent methods to my code.

State Jump Table

In "ENEE 440 - PIC18 Alarm Clock Main.asm" line 1064. I wanted a constant time function to get to a certain state. We could compare to zero, subtract 1, compare to zero… (can do the same thing with XOR) but the last state on the list would not be reached for many step. What I did was add the state value to the PC (see page 23 for more detail about this) and at the address the PC ended up at, would be a GOTO to the correct state. The downside to this, is that we take up program memory with GOTO's that will never be reached (I have pointed them to the default state in case of error).

Automatically Switching State

By setting up a counter that decrements every 1/100 of a second, we have pretty good and easy control of switching states every time main loop is called without taking too much extra time (see page 17 for more detail). The actual structure yields checking one register to see if it is negative to tell if the counter is active or not.

Responding to Pressing of a Switch

This is fairly well documented in the Switches section on page 12. But without too much code we can respond to a switch being pressed, pressed and released, or held down. We have a 'hold down limit' that allows us to easily change how long it takes to consider a switch 'held down.' (from zero to 1.27 seconds, though this needed to be changed to 0 to 32767 for the quicker demo file, since 1.27 seconds passed hundreds of times quicker; thus requiring an extra register).

Appendices

Appendix A: Acronyms and Abbreviations

Acronym/ Abbreviation	Meaning/ Translation
7Seg	Seven Segment (Display)
BCD	Binary Coded Decimal
JSCII	John's Standard Code for Information Interchange
LED	Light Emitting Diode
RE	Rotary Encoder
SW	Switch
hh	two nibbles of hours in BCD format
mm	two nibbles of minutes in BCD format
ss	two nibbles of seconds in BCD format
cc	two nibbles of hundredths of seconds in BCD format
MT	MASTER_TIME
Md	MASTER_DATE
MD	MASTER_DAY

Appendix B: Global and Important Variables

I used 45 out of the 96 allowed UDATA registers (the first 0x59 slots of ram). In the code there are some UDATA labels that are old (especially in the switch file). Besides the temporary ones, the 'in use' ones are in tables below.

MASTER_STATE		
8-bits		
A	S/N	State
Holds the current state. A = 1 when alarm is ringing, S/N = 1 when in snooze/nap mode.		
SET_MS (macro), Main Loop, most states		
ENEE 440 - PIC18 Alarm Clock Main.asm		

MASTER_TIME			
32-bits			
hh	mm	ss	cc
Keeps track of current time in hours, minutes, seconds and 1/100 of a second			
Set_Snooze, Alarm_Ringing, Tick_MT, others			
ENEE 440 - PIC18 Alarm Clock MT.asm			

MASTER_DATE
16-bits
4 digit date in compact BCD form
Keeps the current date
Set_Snooze, Alarm_Ringing, Tick_MT, others
ENEE 440 - PIC18 Alarm Clock MT.asm

MASTER_DAY							
8-bits							
-	s	F	R	W	T	M	S
Holds the day of the week, only one bit set at once, rotate left to get next day of the week, if negative, rotate again.							
States 21 and 31							
ENEE 440 - PIC18 Alarm Clock MT.asm							

In_Service_L							
8-bits							
high							low
Bits tell which sub-priority, low interrupts are in service, highest sub-priority in bit 7. 1 = in service 0 = not in service							
LowInt							
ENEE 440 - PIC18 Alarm Clock Low Int.asm							

Start_Srvs_L							
8-bits							
high							low
Bits tell which low sub-priority interrupt needs to be started. Highest sub-priority is bit 7 1 = needs service 0 = does not need service							
LowInt							
ENEE 440 - PIC18 Alarm Clock Low Int.asm							

DelayCount
16-bits
2's compliment number
Holds the number of 0.01 seconds left until we need to change to the state in DelayState. If negative, it means it should not be decremented. Delay range: [10 ms, 5.46 min]
Time_Change_State, Many states
ENEE 440 - PIC18 Alarm Clock Main.asm

DelayState		
8-bits		
-	-	*state*
Hold the number of a state, when DelayCount becomes zero, we should set MASTER_STATE to this register's value		
Time_Change_State, Many states		
ENEE 440 - PIC18 Alarm Clock Main.asm		

HDCount
8-bits
2's compliment number
Similar to DelayCount, how many 0.01 seconds left until a switch has been 'held down' as opposed to pressed and released.

When negative, it is disabled.
Many States
ENEE 440 - PIC18 Alarm Clock Main.asm

Alarm_SP
16-bits
RAM address
Keeps track of the top of the alarm stack
AlarmRinging_q, AddAlarm, AddAlarmSpace, ChangeCurAlarm, DeleteAlarm, Update-ALARM_STATE, qSpecial_Alarm_exist_q
ENEE 440 - PIC18 Alarm Clock Alarms.asm

Alarm_mod_SP
16-bits
RAM address
Keeps track of the top of the alarm stack
AddAlarm, AddAlarmSpace, ChangeCurAlarm, DeleteAlarm, UpdateALARM_STATE, qSpecial_Alarm_exist_q, AlarmReset
ENEE 440 - PIC18 Alarm Clock Alarms.asm

snooze_duration	
8-bits	
hh	mm
Holds the default length of snoozing after an alarm rings in hours and minutes (compact BCD)	
AlarmReset, Set_Snooze	
ENEE 440 - PIC18 Alarm Clock Alarms.asm	

nap_duration	
8-bits	
hh	mm
Holds the default length of naps in hours and minutes (compact BCD)	
AlarmReset, Set_Nap	
ENEE 440 - PIC18 Alarm Clock Alarms.asm	

Display_Buf							
64-bits							
LM 7Seg	LC 7Seg	: , .	RC 7Seg	RM 7Seg	GRN LED	-	RED LED
Contains direct translation of what should be displayed on the LED's.							
Refresh_LED, LED_Write							
ENEE 440 - PIC18 Alarm Clock Main.asm							

LED_cnt							
8-bits							
1	*1*	*1*	*0*	*1*	*1*	*1*	*1*
Used to both drive the Anodes and also pick which byte of Display_Buf to refresh to the screen (there are 8 pairs). What is shown above is the contents at one point in time, next time the function is called, that will be rotated.							
Refresh_LED							
ENEE 440 - PIC18 Alarm Clock LED.asm							

RE_POS
16-bits
2's Compliment
Holds the position of the rotary encoder, rotating clockwise corresponds to increasing this number. This number will be incremented by 96 for every full revolution.
RE_intrpt, snooze_manage
ENEE 440 - PIC18 Alarm Clock RE.asm

RE_state							
8-bits							
a	b	-	-	-	-	-	-
Previous state of the rotary encoder. Compare to current state to see if a change, increase or decrease RE_POS. 01 -> 00 : clockwise 01 -> 11 : counterclockwise 11 -> 01 : clockwise 11 -> 10 : counterclockwise just half of the changes possible.							
RE_intrpt							
ENEE 440 - PIC18 Alarm Clock RE.asm							

SwTmpCmp							
8-bits							
Used for a variety of switch tasks, from holding values to compare to just temporary storage.							
QuerySwitchesII, UpdateSwitches							
ENEE 440 - PIC18 Alarm Clock SW.asm							

Sw_State_16							
8-bits							
-	P/n	P/n	P/n	P/n	P/n	P/n	-
Updated on an interrupt to keep the bits consistent with which switches are currently being pressed. Bit 1 is switch 1… bit 6 is switch 6. Accurate to two switches being pressed at once. 1 = Currently Pressed 0 = Currently Depressed Once a switch is read by a function, that function may (and should) clear that bit to say that the switch press has been registered.							
UpdateSwitches, whoever need to know which switches are currently pressed							
ENEE 440 - PIC18 Alarm Clock SW.asm							

Sw_State_7C							
8-bits							
-	P/n	P/n	P/n	P/n	P/n	P/n	-
Same as Sw_State_16, except bit 1 is switch 7… bit 6 is switch C (add 6 to the bit values to get the corresponding switch value)							
UpdateSwitches, whoever need to know which switches are currently pressed							
ENEE 440 - PIC18 Alarm Clock SW.asm							

Sw_PState_16							
8-bits							
-	P/n	P/n	P/n	P/n	P/n	P/n	-
Before Sw_State_16 is updated, it should be copied here so functions can see which switches were recently released.							
UpdateSwitches, whoever need to know which switches were pressed previously							
ENEE 440 - PIC18 Alarm Clock SW.asm							

Sw_PState_7C							
8-bits							
-	P/n	P/n	P/n	P/n	P/n	P/n	-
Before Sw_State_7C is updated, it should be copied here so functions can see which switches were recently released.							
UpdateSwitches, whoever need to know which switches were pressed previously							
ENEE 440 - PIC18 Alarm Clock SW.asm							

Appendix C: Function API

Main	
Arguments	
Return Value	
Description	Contains main loop, initiates registers and interrupts
Special Notes	
Variables Used	MASTER_STATE
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

Manage_Alarms	
Arguments	
Return Value	
Description	Checks to see if an alarm is ringing and responds accordingly if one is ringing (includes responding to user input to turn it off)
Special Notes	Self serving, externally, only alters MASTER_STATE
Variables Used	MASTER_STATE (MASTER_TIME/DATE alarm stack)
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

Manage_Snooze	
Arguments	
Return Value	
Description	Displays the time left of your snooze session (the difference between the current time and the snooze alarm's time), responds to the rotary encoder by updating the snooze alarm, as well as exiting if nap is ended by rotary encoder.
Special Notes	Regular states not arrived at while snooze bit is set
Variables Used	MASTER_STATE, alarm stack (MASTER_TIME/DATE)
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

Time_Change_State	
Arguments	
Return Value	
Description	Changes MASTER_STATE to DelayState when DelayCount is zero, reset DelayCount
Special Notes	
Variables Used	MASTER_STATE, DelayCount (DelayState)
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

QreturnToState00Q	
Arguments	
Return Value	
Description	Looks at the switched currently pressed, if 't' is pressed, then go to state30, set up delay state to take you to the default state in 0.2 seconds
Special Notes	Go to state30 first so pressing 't' not carried over to State00
Variables Used	Sw_State_7C, MASTER_STATE, delayCount, delayState
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

QdeleteAlarmQ	
Arguments	
Return Value	
Description	Similar to QreturnToState00Q, except looks at 'd' to see if it has been held down. If so, go to state0D on a timer, then to state1D.
Special Notes	Should only be called from Review Alarm states (states in side the larger circle in Figure 6.
Variables Used	Sw_State_7C, MASTER_STATE, delayCount, delayState
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

QchangeCurrentAlarmQ	
Arguments	
Return Value	
Description	Same as QdeleteAlarmQ, but change FSR1, FSR2 to point to the next alarm in the stack (or first if we go over)
Special Notes	Should only be called from Review Alarm states (states in side the larger circle in Figure 6.
Variables Used	Sw_State_7C, MASTER_STATE, delayCount, delayState
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

YesOrNo	
Arguments	
Return Value	WREG: -1 nothing pressed, 0 = no selected, 1 = yes selected
Description	Displays yes or no. Calls LED_Write as a subroutine
Special Notes	Calling state should not try to write to the LED buffer
Variables Used	Sw_State_16
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

DisplayTest_A	
Arguments	
Return Value	
Description	Displays a variety of items, (uncomment what you want displayed) Calls LED_Write
Special Notes	For testing only (not used in final product)
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

DisplayTest_B	
Arguments	
Return Value	
Description	Displays which state you are in when called. Calls LED_Write
Special Notes	For testing only (not used in final product)
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

SetHDCount	
Arguments	HOLDDOWNLIMIT
Return Value	
Description	Checks to see that HOLDDOWNLIMIT is in the bounds of HDCount. HDCount = MIN(HOLDDOWNLIMIT, 0x7F)
Special Notes	
Variables Used	HDCount
Source File	ENEE 440 - PIC18 Alarm Clock Main.asm

AlarmRinging_q	
Arguments	
Return Value	WREG = -1 if not ringing, 0 if ringing
Description	In the interval MT[sec] = 00, MT[hun] < 10, check for alarms: alarm date = Md, alarm time = MT[hh:mm], if so return positive
Special Notes	Keeps FSR0 the way it found it
Variables Used	MASTER_DATE, MASTER_TIME, FSR0
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

silenceAlarm	
Arguments	WREG lower byte of pointer to alarm stack
Return Value	
Description	Look at alarm pointed to by WREG's modification code, and apply it to the alarm (alter or not the alarm's date in the stack).
Special Notes	
Variables Used	FSR0
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

AddAlarm	
Arguments	
Return Value	
Description	Creates an 'empty' alarm on the stack. Does not alter the stack pointers though (this way we do not have to save the alarm until it is 'set') Modification data set up so that the code is 0xFF (we do not know if we repeat or not).
Special Notes	Assumes caller function has checked to see that the stack is NOT full.
Variables Used	Alarm_SP, Alarm_mod_SP, MASTER_DATE, MASTER_TIME
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

AddAlarmSpace	
Arguments	
Return Value	
Description	Changes the stack pointers to add an alarm onto the stack (in the way I have used it, I add the alarm first (stack does not recognized it yet) and then I update the stack pointer when I print 'SET' on the display.
Special Notes	Assumes caller function has checked to see that the stack is NOT full.
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

ChangeCurAlarm	
Arguments	
Return Value	
Description	Changes the stack pointers, FSR1 and FSR2 to point to the next alarm in the stack (or rotate around)
Special Notes	Should only be called from Review Alarm states (states in side the larger circle in Figure 6.
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

DeleteAlarm	
Arguments	
Return Value	
Description	Deletes current alarm pointed to by FSR1 (and moves the rest of the alarms in the stack down one space)
Special Notes	Should only be called from Review Alarm states (states in side the larger circle in Figure 6.
Variables Used	FSR1, FSR2
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

qSpecial_Alarm_exist_q	
Arguments	
Return Value	WREG: -1 not special alarms exits, 0 some exist
Description	Checks the alarm stack pointers to see if an alarm exists
Special Notes	
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

qSpecial_Alarm_full_q	
Arguments	
Return Value	WREG: -1 stack NOT full, 0 stack is full
Description	Similar to last, sees if the stack is full or not.
Special Notes	
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

AlarmReset	
Arguments	
Return Value	
Description	Sets up the alarms (mainly the stack and default values for the daily and snooze alarm as well as their modifications).
Special Notes	
Variables Used	Alarm_SP, Alarm_mod_SP, snooze_duration, nap_duration
Source File	ENEE 440 - PIC18 Alarm Clock Alarms.asm

Refresh_LED	
Arguments	
Return Value	
Description	Copies values in Display_Buf to display (one area at time) also checks to see if the daily alarm will ring (looking at ALARM_STATE, not figuring it out here)
Special Notes	
Variables Used	ALARM_STATE, LED_cnt, Display_Buf
Source File	ENEE 440 - PIC18 Alarm Clock LED.asm

Reset_LED	
Arguments	
Return Value	
Description	Sets up default values for the LED displays as well as the welcome screen.
Special Notes	
Variables Used	LED_cnt, Display_Buf
Source File	ENEE 440 - PIC18 Alarm Clock LED.asm

BCD_to_7SEG	
Arguments	WREG (JSCII offset code)
Return Value	WREG (translated code)
Description	Uses WREG as an offset to the JSCII lookup table
Special Notes	Does not check to see that WREG is within table limits
Variables Used	TBLPTRL/H/U
Source File	ENEE 440 - PIC18 Alarm Clock LED.asm

LED_Write	
Arguments	WREG, FSR0
Return Value	
Description	Writes values to the display buffer. Value is in WREG, if FSR0H is 1, then translate this value using BCD_to_7SEG, else just write directly. FSR0L indicates which parts of the display buffer should be written to.
Special Notes	
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock LED.asm

Tick_MT	
Arguments	
Return Value	
Description	Called from interrupt, should be called every 1/100 of a second, works with Timer 1's counter to accomplish this. Increments MT, Md and MD by 1/100 of a second, also decrements DelayCount and HDCount if they are not turned off (>0). Takes care of flashing the : at 1 or 5 Hz depending on the state. Also flashed the LED if the alarm is currently ringing.
Special Notes	MT, Md, MD, DelayCount, HDCount
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

Add_MT	
Arguments	WREG (in 1/100 of a second, compact BCD form)
Return Value	
Description	MT = MT + WREG
Special Notes	
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

Reset_MT	
Arguments	
Return Value	
Description	Initializes registers, gives initial time, should not be on the first day (Md != 0000).
Special Notes	
Variables Used	MASTER_TIME, MASTER_DATE, MASTER_DAY, TMR1
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

Display_[day of the week]	
Arguments	
Return Value	
Description	Writes the day of the week to Display_Buf
Special Notes	[day of the week] = Sun, Mon, Tue, Wed, Thu, Fri, or Sat
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

MT_MD_Change_q	
Arguments	FSR0
Return Value	
Description	Uses the arrow keys to update certain registers, treating them like hours, dates, minutes (seconds). Based on FSR0 we can alter many different registers: 1 MT[mm::ss] 2 MT[hh:mm] 4 Md[dddd] 8 Daily Alarm[hhmm] 9 Snooze defaults [hh:mm] A nap defaults [hh:mm] B Current alarm [hh:mm] C Current alarm [dddd]
Special Notes	A little complicated, but it works.
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

MD_Change_q	
Arguments	
Return Value	
Description	Similar to the previous function, but only changes the day of the week, (the current day of the week).
Special Notes	
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

MTMD_dq_IncDec_Date_High	
Arguments	WREG (compact BCD form)
Return Value	
Description	Register pointed to by FSR0 treated like the higher byte of a date. FSR0* = WREG + FSR0*
Special Notes	WREG = 1 0x01 10 0x10 -1 0x99 -10 0x90
Variables Used	FSR0L
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

MTMD_dq_IncDec_Date_Low	
Arguments	WREG (compact BCD form)
Return Value	
Description	Register pointed to by FSR0 treated like the lower byte of a date. FSR0* = WREG + FSR0*
Special Notes	See special notes in MTMD_dq_IncDec_Date_High
Variables Used	FSR0L
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

MTMD_dq_IncDec_Hour	
Arguments	WREG (compact BCD form)
Return Value	
Description	Register pointed to by FSR0 treated like in hour format. FSR0* = WREG + FSR0*
Special Notes	See special notes in MTMD_dq_IncDec_Date_High
Variables Used	FSR0L
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

MTMD_dq_IncDec_MnSc	
Arguments	WREG (compact BCD form)
Return Value	
Description	Register pointed to by FSR0 treated like a minute or a second (both behave the same) FSR0* = WREG + FSR0*
Special Notes	See special notes in MTMD_dq_IncDec_Date_High
Variables Used	FSR0L
Source File	ENEE 440 - PIC18 Alarm Clock MT.asm

RE_intrpt	
Arguments	
Return Value	
Description	Compares the current state of the rotary encoder with the previous (recorded) state. Adjusts RE_POS higher if rotated clockwise, lower if counterclockwise. See page 17.
Special Notes	Should call on interrupt faster then a person can spin the dial.
Variables Used	RE_state, RE_POS
Source File	ENEE 440 - PIC18 Alarm Clock RE.asm

RE_reset	
Arguments	
Return Value	
Description	Clears Rotary encoder registers
Special Notes	
Variables Used	RE_state, RE_POS
Source File	ENEE 440 - PIC18 Alarm Clock RE.asm

QuerySwitchesII	
Arguments	
Return Value	WREG (nibble = no switch pressed, nibble not zero means that switch is pressed)
Description	Checks the SPP port to see which switches are pressed. First checks to see if any are pressed, if so we allow the rightmost four switches to register responses, this response is unique for each combination of those four switches being pressed, using this we can see which switches are pressed. Continue to next four switches. We stop when two switches are identified (even if there are more pressed). The rightmost ones are identified first.
Special Notes	Uses FSR0, identifies up to two switches being pressed at once.
Variables Used	SwTmpCmp,
Source File	ENEE 440 - PIC18 Alarm Clock SW.asm

UpdateSwitches	
Arguments	
Return Value	
Description	Calls QuerySwitchesII, updates Sw_State_16, Sw_PState_16, Sw_State_7C, Sw_PState_7C with current (previous) switch states.
Special Notes	Should call from interrupt slow then the period of bouncing.
Variables Used	ENEE 440 - PIC18 Alarm Clock SW.asm
Source File	ENEE 440 - PIC18 Alarm Clock SW.asm

resetSwictches	
Arguments	
Return Value	
Description	Looking back, this is not needed, since we don't use US_pot.., or Sw_State_HD/PR anymore.
Special Notes	(Those variables were part of a failed attempt to monitor the switches that would have taken most of the switch management out of the individual states).
Variables Used	
Source File	ENEE 440 - PIC18 Alarm Clock SW.asm

Appendix D: Macros

MOVLW	
Arguments	literal
	file register (8-bit address)
Result	REG = literal

SET_MS	
Arguments	newState
Result	MASTER_STATE[5..0] = newState[5..0]

SET_DS	
Arguments	nextState
	delayTime (16 bytes, between 0 and 32767) time is in 1/100 of a second
Result	DelayState = nextState DelayCount = delayTime

CLR_DS	
Result	DelayState is default state DelayCount is disabled (DelayCount[15] = 1)

TmrIntrpt_setup	
Arguments	Timer (0 through 3)
	eight_sixteen (alarm specific)
	prescale (alarm specific)
	interrupt (1 for yes, 0 for no)
	priority (1 for high interrupt, 0 for low)
	clear (if we want to clear the counter)
Result	Read the code for more details, but sets up one of four timers as potential interrupts.

MOVWSPP	
Arguments	WREG (value to write)
	address (address to write to)
Result	SPPEPS = address SPPDATA = WREG Safety (integrity) check performed to assure proper writing of value

MOVSPPW	
Arguments	address
Result	WREG = value at that address. Safety (integrity) check performed to assure proper reading of value

JSCII_TO_DISPLAY	
Arguments	area (which 7Seg to write to)
	jscii (hex offset in JSCII table)
Result	offset translated and stored at correct area in display buffer.

SEGMENTS_TO_DISPLAY	
Arguments	area (which 7Seg to write to)
	jscii (actual bits to be displayed)
Result	Display_Buf[area] = jscii

Sub_P_set_L	
Arguments	flag_reg (register where interrupt flag is)
	flag_bit (bit in register which is the flag)
	s_p (sub-priority: 0-7)
Result	Start_Srvs_L[s_p] is 1 if service is needed flag_reg[flag_bit] is cleared (turn off intrpt)

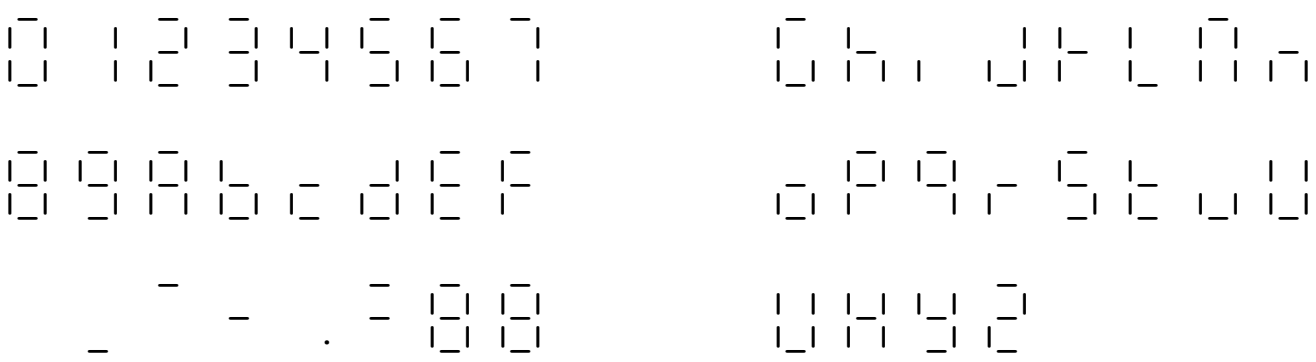
Sub_P_ex_L	
Arguments	where_to (Label, **Call** to start service)
	s_p (sub-priority: 0-7)
Result	Write in order of priority (highest first). If currently executing, will continue to execute interrupt, if we need to start, start. Clears corresponding Start_Srvs_L bit when we start as well setting the In_Service_L bit.

Appendix E: JSCII Table

This is a set of characters mapped to program memory and are indexed by their hex value, similar to how ASCII works. The first 16 entries line up to the numbers 0 through F, so translation of hex and BCD numbers has been efficienized.

Character	Offset	Character	Offset	Character	Offset
JSCII_0	0x00	JSCII_F	0x0F	JSCII_M	0x26
JSCII_1	0x01	JSCII_SP	0x10	JSCII_N	0x27
JSCII_2	0x02	JSCII_UN	0x11	JSCII_O	0x28
JSCII_3	0x03	JSCII_UP	0x12	JSCII_P	0x29
JSCII_4	0x04	JSCII_HY	0x13	JSCII_Q	0x2A
JSCII_5	0x05	JSCII_DP	0x14	JSCII_R	0x2B
JSCII_6	0x06	JSCII_EQ	0x15	JSCII_S	0x2C
JSCII_7	0x07	JSCII_CO	0x16	JSCII_T	0x2D
JSCII_8	0x08	JSCII_AP	0x16	JSCII_U	0x2E
JSCII_9	0x09	JSCII_G	0x20	JSCII_V	0x2F
JSCII_A	0x0A	JSCII_H	0x21	JSCII_W	0x30
JSCII_B	0x0B	JSCII_I	0x22	JSCII_X	0x31
JSCII_C	0x0C	JSCII_J	0x23	JSCII_Y	0x32
JSCII_D	0x0D	JSCII_K	0x24	JSCII_Z	0x33
JSCII_E	0x0E	JSCII_L	0x25		

How they appear on the 7Seg display (W and M should repeat themselves)



Appendix F: Other Constants

SPP Address Definitions

CATHODES	0x00
ANODES	0x01
SWITCHES	0x02
RE_ENC	0x02

Rotary Encoder Constants

RE_Bit_A	7
RE_Bit_B	6
RE_Mask	b'11000000'

LED Segment Definitions (Cathodes)

SS_A	b'11011111'
SS_B	b'11110111'
SS_C	b'10111111'
SS_D	b'11111101'
SS_E	b'11111011'
SS_F	b'01111111'
SS_G	b'11101111'
SS_P	b'11111110'

SS_ON	b'00000000'
SS_OFF	b'11111111'

SS_apo	b'10111111'
SS_col	b'11010111'

LED_11	b'01111111'
LED_12	b'10111111'
LED_21	b'11011111'
LED_22	b'11111011'
LED_31	b'11111110'
LED_32	b'11111101'

LED Segment Bit Definitions (Cathodes)

LED_11_bp	7	LED bit position
LED_12_bp	6	LED bit position
LED_21_bp	5	LED bit position
LED_22_bp	2	LED bit position
LED_31_bp	0	LED bit position

LED Area Definitions (Anodes)

DISP_RM	b'11110111'	rightmost 7seg
DISP_RC	b'11101111'	rightcenter 7seg
DISP_LC	b'10111111'	leftcenter 7seg
DISP_LM	b'01111111'	leftmost 7seg
DISP_COL	b'11011111'	colon and apostraphy
DISP_RED	b'11111011'	red LED's
DISP_GRN	b'11111110'	green LED's
DISP_YLW	b'11111010'	both red and green
DISP_ALL	b'00000000'	all on
DISP_7SEG	b'00100111'	all 7segs on
DISP_NONE	b'11111111'	all off

To set the high bit of FSR0 in LED_Write argument to translate the WREG argument

DISP_Trans b'100000000'

Alarm Stack Constants

Alarm_Base_addr	0x100
Alarm_Base_size	0x80
Alarm_Mod_Base_addr	0x180
Alarm_Mod_Base_size	0x80

Timing (in 1/100 of a second)

HOLDDOWNLIMIT	0x30	480 ms
STD_Times_Out	0x0320	8 seconds
STD_GoBack_Wait	0x0190	4 seconds
STD_Disp_Wait	0x00C8	2 seconds

Appendix G: Developer's Notes

These are note I collected and typed out (in addition to paper notes, which will not be included). They are an insight into my successes and frustrations. They are not complete, and should be taken at face value.

4-4-2008
Displayed "John [traffic light]" on board using display algorithm.

4-5-2008
Slowed down display refresh rate to ~1Hz, as well as spaced our 'ghosting prevention step' from refresh step to assure that ghosting was not a problem.

4-6-2008
When writing values to display using LED_Write rightmost seven segment value shown on leftmost seven segment display as well, when calling left to right. Fixed by writing left to write, but leftmost last. Not a problem with definitions. Not an issue with Refresh_LED subroutine. Possibly issue with writing to Display_Buf + n ?

4-7-2008
Query switches successfully and completely works (for up to two switches pressed in different blocks).

4-8-2008
Master time interrupt is working (and accurate to 100Hz), timer 0 (and others) are causing ghosting and incorrect display issues. One of which was a previous test I left in.

4-10-2008
Found source of error talked about on 4-6-2008, had a test 'I was here' that was causing Display_Buf + 0 to get value of Display_Buf + n's value when written to by my function LED_Write.

4-11-2008
Can't get switches to work with HD and PR states. Have opted to just update flags in a register when switches are active.
See notes on how to use switches in 'switch' section. Interrupt controls press and release of switch if any function that looks at pressing a switch clears that switches flag. Interrupting at ~45.7Hz, too fast, increments at about 34 Hz. Can press and release and increment just once, but that is hard to do. Cut interrupt frequency in half: on 'normal push' increments by 2. Cut in half again, now it a 'normal push' sometimes doesn't increment at all. I'll leave it cut in half only once: making it interrupt at about 23 Hz. Function will have to manage it smarter.
Switch hold down counting seems to work well, EXCETP: causes red and green LED's to change. Problem with any switch that I hold down when associated with my test code. Without test code, (or testing with that switch) you can hold down any switch and LED's stay they same. Also seems to change the 7seg display too. Error doesn't happen if we don't call LED_Write. Doesn't seem to be a problem with LED_Refresh subroutine either. *Usually* changes display of all digits (not just 1, or 1 color). Places getting written to often (in main loop) seem to fix themselves (see a flicker, but stays correct). So an issue with Display_Buf's integrity? Doesn't seem to turn on lower left LED light.
This error only happens when I hold down the button long enough to increment the test counter. Seems to effect the LED's more then the 7seg. Also seem to always be numbers displayed, not random lights. Seems to pick one of the nibbles (usually lower) in MAIN_TEST counter and write it to some of the spaces in Display_Buf (after translation). Sometimes error display is MSnibble of counter, but not always. Seems like LED_Write is being called randomly with random arguments.

Not an issue with SetHDCount. Was accidentally disabling HDCounter each time we hit zero, changed that, errors seem to happen less frequently. Works when these two lines are commented out:

```
CALL  SetHDCount            ;; reset counter
SETF  HDCount, 0            ;; since let go, disable counter
```

I don't know why (acts like PR test code).
AAAAHHHHH!!! I found the error, in MT call, I was DECFSZ instead of DECF, skipping backing up the FSR0L register, which is part of the display routine. Too many hours used to figure that out. (those two lines enabled HDCount to be decremented inside of MT call).

4-24-2008

After setting alarm to ring (MS[7]) it seems to get turned off automatically, not sure where. If you set it to ring in your main loop, it 'rings' properly (LEDs flash green, yellow, red, off at 1 Hz). Also, states 26 and 36 don't proceed properly, have to hit 't' to get out of, then are stuck, can't move buttons for a couple seconds. (States 06 and 16 seem to still work).

4-25-2008

Not an error: got time/date change functions to work off of pointers (call with amount change in W, and what to change in FSR0). Makes minutes/seconds the same code, is really just the rules on the range of the value you want to change (minutes can only be between 00 and 59).

4-26-2008

Cannot have alarms at 0x800 (restricted). Putting them at 0x100, also placing alarm modifications at 0x180, so you will only be able to have 30 custom alarms (as opposed to 64).

LED's do not refresh correctly when rotary encode interrupt is established. I wasn't backing up FSR0, did that and now I can see what should be displayed, except lots of flickering and buttons are not the same anymore. Removal of earlier attempt to correct display corrected button use, but still flicker. Writing 1's to the anodes fixed this. Ghosting still seems to exist, but I don't think it is from the rotary encoder.

Strange rotary encoder error: when turning from RE_POS > 0 to RE_POS < 0, we turn on the bottom two LED's to green (depending on circumstance, turns on all green LED's). Ok, not always turning on the same green LED's, but most of them, most of the time.

4-30-2008

Had an error when turning on the 'buzz' interrupt: display was flickery, adjusted the SPP macros, then there was no display update. Just undid it all.

5-1-2008

Got the dial in the snooze to work, but most of the snooze part does not. Calculation of snooze alarm from MT/Md and snooze_duration is not working correctly. I think it comes from DelayState being altered in state00, which needs to be partially run to see display, but the button pressing part should be skipped over. Seemed to have fixed that problem ☺. After I took out the code to test the display (made it look at switches in state00) it stared flickering a bunch, and only in state00, everything else seems to work. (Pressing 'a' while alarm is ringing still goes to snooze).

Tried putting display refresh in RE interrupt, but display is very dim, though should be at around 2kHz. No real flicker though. Tried in MT interrupt (100Hz) very, very flickery, seizer. Copied the call to Refresh_LED in the middle of State00, seemed to work well, well, maybe not: left and right most 7seg look bright, but others duller. Sometimes it switches, and the others are brighter (seems to depend on MT[min, low]). I put it in twice (into MS00) and it seems to help, but other states brighter.

Have to change daily alarm's date when you change master date.

5-2-2008

Having trouble showing the difference between time of alarm and master time (difference is how long is left of snooze alarm). Adjusting for BCD and 0 through 59 possible is too hard, so subtract both, if result gives no borrow (STATUS[0]) then ok, else minutes left is 60-Alarm's Minutes + Time's Minutes.

5-4-2008

Starting to add the code to add/modify user defined alarms. Yesterday and today I changed how you review/set alarms, it is all done in the same functions, just start and exit differently. Don't let modify stack pointers until done editing alarm, then you don't have to worry about it ringing while setting it. Also allows you to cancel that alarm (not written till you see 'set')

5-5-2008

Trying to get programming done tonight (won't happen) seem to add alarms that ring everyday or once (won't get certain days of the week). Still need to have LED's reflect what you are doing.

5-6-2008

Able to successfully add and delete alarms. Made the decision to not implement setting (or seeing) an alarm to the day of the week (can't have it ring every Tuesday), but have left in the states to do so, just not filled up (taken care so we don't accidentally get to one).

Fixed RE error of turning down past 00 00 to FF FF, there was a DECFSZ instruction that should have been a DECF.

Almost done, need to:
If daily alarm rings, set 'ALARM ON' bit to red, if special alarms set that LED to green.
Need to make the modifications work correctly.
need to set/unset daily alarm

5-7-2008

Got 'Alarm On' light to properly light.

To set/disable the daily alarm:

disable: set date to 00 00

enable: set date to Md, check to see if set, if not, add 1

D10K

I have NOT done much/any testing of the alarm clock when the date roles over from 9999 to 0000. I know I have stated that the initial time should not be in the first day, mostly because that is where I place the daily alarm when it is not ringing. The error could range from the daily alarm going off no matter what on day 0000 to the user being confused for various other reasons.

Index

T