

```

1  init: 12V Supply          35  symbol setupmins = b1          66  let fanenable = 1
2
3  pause 500
4  serout 6,N2400,(254,1)
5  pause 2000
6  i2cslow,
7  i2cbyte
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97

symbol setuphour = b2
symbol setupday = b3
symbol setupdate = b4
symbol setupmonth = b5
symbol setupyear = b6

symbol setupalarm = b7
symbol setupalarm = b8
symbol setupfull = b9
symbol setupfade = b10
symbol setupdism = b11

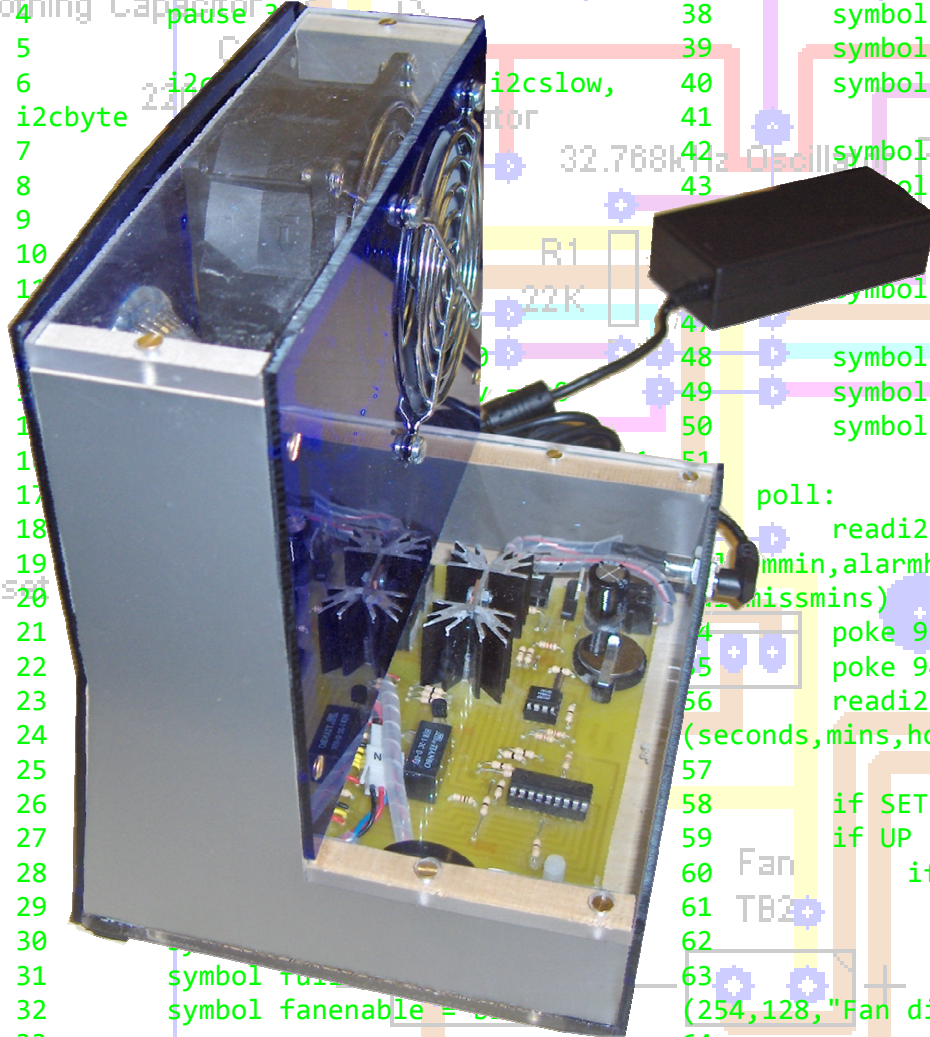
symbol digit = b12
symbol value = b13
symbol maxvalue = b14

poll:
  readi2c $08,
  alarmmin,alarmhour,fullweek,dismissmins)
  poke 93,fademin
  poke 94,dismissmin
  readi2c $00,
  (seconds,mins,hour,day,date,
  57
  58  if SET = 1 then set
  59  if UP = 1 then
  60    if fanenable =
  61      let fanena
  62      low 7
  63      serout 6,N2400,
  64      (254,128,"Fan disabled!")
  65      pause 2000
  66    else
  67      printtime:
  68      gosub lamp
  69      pause 2000
  70      let digit = hour & %00110000 /

```

# f00n DS4 (dreamsystem four)

## David Piggott



# Contents

---

Project Outline	4	Development - BCDs and Programming	57
Time Plan	5	Development - Lamp Dimming	62
Task Analysis	6	Development - Putting it all together (Programming)	63
Research - Existing Products	10	Development - Final Breadboard Program	86
Research - Electronics - Processing	10	Research - Electronics - Input (Control) Components	96
Research - Electronics - Output (Wake-up Call) Components	15	Research - Electronics - Output (Wake-up Call) Components	100
Research - Electronics - Output (Control) Components	17	Research - Electronics - Power Supply Components	102
Specification	20	Production - Schematic 1.0	103
Initial Circuit Ideas - System Diagrams	22	Production - Artwork 1.0	104
Initial Casing Ideas	26	Production - Artwork 1.0 (Colour Coded)	105
Development - Breadboarding & Programming	27	Production - Schematic 2.0	106
Development - Progress Review	48	Production - Artwork 2.0	107
Development - A Programming Revelation	49	Production - Artwork 2.0 (Colour Coded)	108
Development - False Assumptions	52	Production - PCB 1.0 from Artwork 2.0	109



# Contents

---

Production - Testing PCB 1.0	110	Case Development - Side Profile	122
Production - Schematic 3.0	111	Case Production	123
Production - Artwork 3.0	112	Final Program - Cutting down the bits	127
Production - Artwork 3.0 (Colour Coded)	113	Final Program	133
Production - PCB 2.0 from Artwork 3.0	114	Finished Product - Pictures	144
Production - Testing PCB 2.0	115	Time Management - Diary	146
Production - Schematic 4.0	116	Evaluation & Testing	148
Production - Artwork 4.0	117	Industrial Practices, Social Issues, Systems & Control	155
Production - Artwork 4.0 (Colour Coded)	118	Bibliography & Conclusion	157
Production - PCB 3.0 from Artwork 4.0	119	Appendix A (Datasheets, Sample Program Thread)	158
Case Development - Design Overview	120	Appendix B (Case Dimensioning Sketches, Program planning)	221

# Project Outline

---

## **What is the aim of the project?**

The aim of the project is to make a specialised alarm clock for heavy sleepers, who cannot rely on typical alarm clocks.

## **What makes the project special?**

I intend to design the alarm so that it is capable of waking even the heaviest of sleepers. However, unlike most alarm clocks aimed at heavy sleepers it will not do so by simply being very loud.

The wake-up call will have two phases. The initial phase will have a gradually increasing intensity to it.

The purpose of this 'soft' mode is to simulate a natural dawn as well as possible. The reason for this is that I believe being slowly woken over a period of ten to twenty minutes is surely better than a sudden disturbance (see below for justification).

The second phase of the wake-up call is a backup of sorts and will be as unpleasant and spontaneous as is necessary to wake the user from their sleep. An example would be a high intensity light flashing on and off.

These principles will be applied to the use of all the output components I equip the alarm clock with.

With regard to timing, I will aim to make as many aspects of the wake-up call as is practical user configurable so that the wake-up call can be tailored to the individual user (for example, the time taken during the first phase for the intensity to reach maximum).

## **Why is a gentle wake up better?**

Generally speaking most people are irritated by being suddenly awoken and instinctively close their eyes. In doing so they drastically increase the risk of falling asleep again.

I believe that this problem can be overcome by slowly increasing the intensity of the components of the wake-up call.

# Time Plan

---

ID	Task Name	Start	Finish	Duration	Jan 2007				Feb 2007				Mar 2007					
					7/1	14/1	21/1	28/1	4/2	11/2	18/2	25/2	4/3	11/3	18/3	25/3		
1	Ongoing paperwork	08/01/2007	20/03/2007	52d	[Blue bar spanning from 08/01/2007 to 20/03/2007]													
2	Breadboarding of circuit ideas	11/01/2007	19/01/2007	7d	[Blue bar from 11/01/2007 to 19/01/2007]													
3	PCBs Artwork Design	19/01/2007	29/01/2007	7d	[Blue bar from 19/01/2007 to 29/01/2007]													
4	PCBs Production & Assembly	29/01/2007	06/02/2007	7d	[Blue bar from 29/01/2007 to 06/02/2007]													
5	PCB Testing	06/02/2007	12/02/2007	5d	[Blue bar from 06/02/2007 to 12/02/2007]													
6	Case ideas development	19/01/2007	20/02/2007	23d	[Blue bar from 19/01/2007 to 20/02/2007]													
7	Case construction	20/02/2007	09/03/2007	14d	[Blue bar from 20/02/2007 to 09/03/2007]													
8	Mounting of electronics within case	01/03/2007	09/03/2007	7d	[Blue bar from 01/03/2007 to 09/03/2007]													
9	Final details & fine tuning of program	12/03/2007	19/03/2007	6d	[Blue bar from 12/03/2007 to 19/03/2007]													
10	Evaluation & Panic Time!	19/03/2007	22/03/2007	4d	[Blue bar from 19/03/2007 to 22/03/2007]													



# Task Analysis

---

- How will the alarm clock wake the user up?
  - The most obvious method is a noise of some sort, as the majority of alarm clocks use.
  - However this is not the only means by which the alarm clock can wake users up; it is an alarm clock for heavy sleepers and thus must have some backup.
  - Keeping in mind that I decided the alarm should start gently and only turn assertive if not dismissed after some time, one ideal output would be a lamp that fades in over a period of several minutes, to simulate sunrise.
  - A lamp that fades in slowly is therefore an ideal way to get around this problem.
  - In addition to a lamp, another possible output is a fan, to simulate a breeze and wake the user up, that gradually increases in intensity in the same way as the lamp.
- How will the real time, alarm time, and other information be displayed?
  - There are three options for this; 7-Seg displays, Starburst displays, or an LCD.
  - The main advantage of 7-Seg displays is their good visibility both in light and dark conditions. However they can only display one numeric character each and require additionally driver circuitry. This would limit the information that I could display with them, and would restrict the options I have when designing the setup and configuration system due to the lack of ability to display text. The driver circuitry would probably require several output pins from the microcontroller for each module, and since I would need a minimum of four just to show the hour and minutes I would most likely exceed the number of output pins available on the microcontroller unless I was to find a serial driver.
  - Starburst displays also offer the advantage of good visibility while also being able to display non-numeric characters. However they require more complicated driver circuitry. Because one display is required for every character I would have to use a great number of them to allow me to create the setup and configuration system I have in mind. They would probably also require a great number of output pins on the microcontroller unless I was to find a serial driver.
  - LCDs do not have the problems that 7-Seg and Starburst displays do; they can print every character I would need and depending on the specific display chosen can be very easy to control with the PICAXE microcontroller system. In addition to this they are multiple character devices, which would enable me to create the setup and configuration system I have in mind (individual, titled menu screens) and display information such as the day, date, month and year.
  - For the reason outlined above the real time, alarm time and all other information will be displayed with an LCD.

# Task Analysis

---

- How many alarm times will the alarm clock support?
  - The alarm clock only needs to support one alarm time. There is no point adding unnecessary complexity in both the development of the product and for the end user in controlling it.
- How will the various settings be changed (real time, alarm time, fade-in time etc.)?
  - A basic system would be one where for every user configurable value there is a separate button; pressing the button increases the value (e.g. the real time hour) until it reaches its maximum (23 in the case of hours) and thereupon loops back to 0. A slight variation on this theme would be the addition of a down button for each value to enable easier decrementing. This system is basic in that the user doesn't depend on feedback from the alarm clock (e.g. knowledge of which setting they have selected for change [i.e. "Setting: Hour"]) other than being able to see what the value actually is (e.g. 23).
  - The other option would be a system where there is only one increment/decrement input pair and this is used for the setting of all the user configurable values (e.g. real time hour, real time minute, real time second... alarm time hour... fade-in time etc.). With this system the user must first select which value they wish to change.
  - For this to work effectively the user must have a means of knowing which configurable value they have selected to change. The ideal way of communicating this to them is by printing the name of the value being edited on the LCD.
  - In order to select the values for changing a minimum of one additional button will be required which I will call the 'set' button.
  - I will explore different ideas for the user interface in greater depth when I create my system diagrams.
- What options will be available during the wake-up call?
  - Most alarm clocks have a snooze and dismiss button. However, I believe the snooze button is only necessary when the user has been woken too quickly. This should not happen with the gentle fade-in of my alarm clock and since pressing the snooze button only results in the user getting up later, I will give the users that option.
  - Choosing whether to include a dismiss button is harder. The main reason for not including one is to prevent abuse, i.e. the temptation to just press the dismiss button and go back to sleep.

# Task Analysis

---

- However, without any means of dismissing the alarm it would just continue with the wake-up call indefinitely. This is clearly a problem; the alarm clock must be dismissed somehow. I have come up with two possible solutions to the problem:
  1. Allowing the user to dismiss the alarm clock but requiring that in order to do so they first get out of bed (and therefore drastically reducing the probability of them going back to sleep again). The simplest way of doing this would be to have the dismiss button separate from the main unit and placed on the other side of the room.
  2. Not allowing the user to dismiss the alarm clock while the wake-up call is running but instead allowing them to set an auto-dismiss time (via one of the setup menu screens). The wake-up call would then be as follows:
    - Wake-up call runs phase one, fading in the lamp and fan over the period specified by the fade-in time setting.
    - After fade-in time has elapsed, wake-up call enters phase two (intensive mode) and runs for the amount of time specified by the auto-dismiss time setting.
    - When auto-dismiss time has elapsed, the wake-up call automatically dismisses.
- In conclusion the alarm clock will have neither snooze nor dismiss buttons and instead will automatically dismiss after a user configurable period.
- How will visibility and the ability to control it in the dark be achieved?
  - The important parts which should be visible in the dark are the control components; that is, the LCD and the setup menu buttons.
  - In order to make the LCD visible in the dark I hope to use a backlit LCD.
  - In order to make the setup menu buttons visible in the dark I could use illuminated buttons, or use standard buttons and place an LED alongside each button.
  - I will need to research both backlit LCDs and illuminated switches.
- Will it be battery or mains powered?
  - The lamp that will simulate dawn will need to have a high maximum intensity. However because it needs to be dimmable I cannot use energy saving lamps as they have a threshold below which they do not operate; thus I am restricted to filament lamps. The problem with filament lamps is that they have quite high current requirements and are less efficient.
  - As such, if I was to make the alarm clock battery powered it would need a very high capacity battery in order to last a reasonable length of time.



# Task Analysis

---

- For this reason the alarm clock will have a mains power supply. However since the electronics will require low Voltage DC power, and for the purposes of safety, I will use a mains power supply that transforms the 230V AC mains supply to 12V DC. This will mean that at no point in the project will I be handling dangerous Voltages.
- Using a mains power supply presents an additional problem. Although I have not researched methods of timekeeping yet, it is fairly obvious that any method of timekeeping requires power to function. This would mean that when the alarm clock is disconnected from the utility power, the time as measured by the clock would cease to advance. Depending on what type of memory I use (volatile or non-volatile) the user-configurable settings (alarm time, fade-in time etc.) could also be forgotten.
- To get around this problem I could have a backup battery that keeps the clock device and memory (if necessary) powered when utility power fails.
- What materials are suitable for the construction of the case?
  - With the aim being to make the alarm clock compact, I do not think wood would be a suitable material due to the additional thickness of it. Visually wood is also inappropriate for an alarm clock.
  - Both metals and plastics could be used for the case. However I do not know enough about the various types available to make a decision at this stage and thus will research this.
- Does it need to be compact and/or portable?
  - Ideally, yes, the alarm clock should be compact and portable. However, the requirement for the lamp and fan will make this a challenging task and already the portability is restricted by my decision to make it mains powered. It will not be possible to have the lamp and fan integrated in the case maintaining compactness.
  - One way to maintain compactness of the main unit would be to make the fan and/or lamp as separate units . However this would be inconvenient for the user because it would require more space and it would add the complexity of connecting the units together and routing the wires between them.
  - To avoid this added complexity for the user I will locate all components in a single unit and strive to make it as compact as possible, though I recognise that I will not be able to make it as compact as if I was to locate the fan and/or lamp in separate units.

## Research - Existing Products

---

I started my existing products research with a search on [www.google.co.uk](http://www.google.co.uk) for “alarm clocks for heavy sleepers”, and found the following products to be most relevant:

### Screamer Alarm

Found at:

[http://www.asseenontv.com/prod-pages/Scremer\\_Alarm.html](http://www.asseenontv.com/prod-pages/Scremer_Alarm.html)

Description:

“Can sleep through the dog barking, the baby screaming, fire engines and trains going past your window, a bomb going off in the other room? Do you feel there is no alarm with enough muscle to wake you up? A trendy version of the original two bell alarm clock it features adjustable alarm control that lets you customize the alarm sound to your own personal needs. The gentle bell for light sleepers, the relatively annoying bell for medium sleepers and the hit the floor running, screaming bell for heavy sleepers. Includes luminous hour and minute hands, pinpoint second hand and night light. Uses 3 "AA" not included. 5"W x 5"H x 4"D.

- Adjustable Alarm Control
- Luminous Hour and Minute Hand
- Pinpoint Second Hand
- Night Light”



### Merits

- Adjustable volume.
- Aesthetically pleasing.
- Luminous parts for visibility in dark.
- Useful night light.

### Weaknesses

- Only uses sound for wake-up call.
- Despite luminescent parts, it could still be hard to read the time in the dark, especially when altering the alarm time.
- Controls are not easily visible in the dark.

## Research - Existing Products

### Merits

- Multiple outputs used for wake-up call— both sound and vibration.
- Very loud alarm cannot fail to wake the sleeper.
- LED Display for easy reading of time in the dark.

### Weaknesses

- Not very aesthetically pleasing.
- Multiple units may inconvenience user when setting it up and after setup due to the space and wiring required.
- Despite LED Display, changing settings (i.e. real time/alarm time) may be difficult because the controls are not illuminated).

### Sonic Boom Clock and Bed Shaker

[http://www.dynamic-living.com/sonic\\_boom\\_clock.htm](http://www.dynamic-living.com/sonic_boom_clock.htm)

“Even the heaviest sleeper will wake up with this extra loud alarm clock! Having trouble hearing the alarm? This loud alarm clock won't let you down. The volume is adjustable up to 110 dB (louder than an average smoke detector!). You can also adjust the tone for a high or low pitch alarm sound.



The Sonic Boom Alarm Clock has an adjustable viewing angle display and 1 inch high bright green LED numbers. It also has a battery back up for power outages (9V battery not included). One year warranty. This loud alarm clock can wake you with the included bed shaker and/or flashing lights - just plug a lamp into the back of the clock. It also has a snooze button.

This Super Shaker 12V Bed Vibrator has a 6 foot cord and is placed under a pillow or between a mattress and box springs. The powerful vibrating bed shaker is guaranteed to wake up extremely heavy sleepers. A built-in temperature sensor protects the unit against overheating.

The Sonic Boom Alarm Clock makes a great gift for school aged children or recent high school graduates that are leaving home to go off to college. For your convenience, the Super Shaker 12V Bed Vibrator is bundled with the Sonic Boom Clock.”





## Research - Existing Products

### Soleil Sunrise Alarm Clock, Dawn Simulator

<http://www.amazon.com/Soleil-Sunrise-Alarm-Clock-Simulator/dp/B0002TISOE>

In nature, sunlight wakes you by triggering your body to produce serotonin, an energizing, awakening type of hormone. Even with eyelids closed, your eyes sense light and signal the production of serotonin. The Soleil Sun Alarm replicates nature - a better way to wake up. The clock's small, sleek, compact design takes up very little space on your night stand. Plus, you can easily pack it in your suitcase or shoulder bag when you travel, so you can enjoy the same gentle, refreshing awakenings when you are away from home. Though it is a physically small clock, it is a very unique, dynamic clock, loaded with many desirable features (see below) that enable you to gently awaken in a relaxed, refreshed state. You will really LOVE this truly amazing, very affective clock. Battery Back-up AM/FM radio Alarm: Alarm feature for waking or timing. Snooze button: 10 minute snooze feature with a 1 minute sunrise, with optional flash and beep at the end of the cycle. Flash & Beeper: Are you a heavy sleeper? These selectable features assist with waking by flashing or beeping at the end of the alarm cycle. Sunrise: Wake with the sun! This feature can be activated at intervals from 15 to 120 minutes, gradually increasing brightness to simulate a sunrise. Sunset: Activate a "sunset" at intervals from 15 to



120 minutes, gradually decreasing brightness of light to simulate a sunset. A great feature for the kids room. Nap Timer: When taking a nap, there's no need to reset the alarm time. Nap timer can be activated at intervals from 15 to 120 minutes. Custom Sunrise: The sunrise feature activates the built-in light which fades in from zero to full intensity over the selected time intervals of 0, 15, 30, 60, 90, or 120 minutes. Security: Automatically turns the lamp on at random intervals to give the impression that someone is at home. Very Compact Size!-measures roughly 5" x 4.5" x 3" UL approved trans-

### General Impression

One of many dawn simulating alarm clocks I found, this is the sort of alternative approach that I have in mind for my alarm clock; the gentle wake-up call.

### Merits

- Multiple outputs used for wake-up call—both sound and light.
- Innovative dawn simulation provides a gentle wake up, reducing the probability that the sleeper simply closes their eyes and goes to sleep again.
- LED Display for easy reading of time in the dark.

### Weaknesses

- Could be more aesthetically pleasing.
- Requires external power supply which inconveniences the user.
- Despite LED Display, changing settings (i.e. real time/alarm time) may be difficult because the controls are not illuminated).

## Research - Electronics - Processing

---

### Requirements

The alarm clock will require a microprocessor to co-ordinate between the control switches, LCD/7-Segs, clock module, and wake-up call devices (fan, lamp, sound output).

The obvious choice for this is the PICAXE range because of the relative ease that programming them is when compared to other PIC microcontrollers. The addition of many project expansion boards further strengthens the attraction of the PICAXE as the microcontroller to use.

The next question is of course which PICAXE is appropriate?

The first deciding factor is the IO (input/output) requirement. Unless I decide to use 7-Seg displays (in which case I would need many output pins), the 18 pin range are ideal, supporting 5 inputs and 8 outputs.

The second deciding factor is memory. The program is likely to be large due to its complexity and therefore would have problems if using a PICAXE with insufficient memory. The X versions have larger memory (eight times as much—600 lines versus 80 according to the PIN Summary datasheet published by Revolution Education).

The third deciding factor is additional features. If I use the Revolution Education LCD module, to be able to read information from the clock module (alarm time, real time), I will need to make the PICAXE interface with it in I2C mode. Only the X versions support I2C. This is explained in greater detail in the development section.

The other requirement is support for the pwmout command (to vary the intensity of the fan and lamp). Again, this is only supported in the X versions.

With all this in mind the logical conclusion is to use the PICAXE 18X.

### PICAXE 18X

Price: £4.35.

Found at <http://www.rev-ed.co.uk/picaxe/>

PICAXE-18X microcontroller chip. Supports 5 inputs and 8 outputs. Extended features include 8x memory and i2c support.



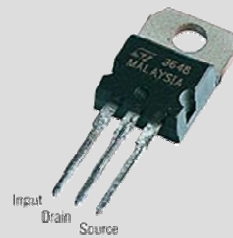
# Research - Electronics - Processing

## Fully autoprotected power MOSFETs

Price: Varies from £0.85 to £2.20 depending on version.

Found at Rapid Online > Electronic Components > Discrete Semiconductors > MOSFETs > Fully autoprotected power MOSFETs.

Fully autoprotected power MOSFETs, intended as replacements for standard power MOSFETs in DC to 50kHz applications. Built-in thermal shutdown, linear current limitation and overvoltage clamp protect the device in harsh environments. Housed in a standard TO-220 package.



- Short circuit protection
- Logic level input threshold
- ESD protection
- High noise immunity
- Schmitt trigger on input
- Low current drawn from input pin

Device VCLAMP RDS(on) Ilim Ptot  
 VNP7N04 42V 0.140 7A 31W  
 VNP10N06 60V 0.30 10A 42W  
 VNP10N07 70V 0.10 10A 50W  
 VNP20N07 70V 0.050 20A 83W  
 VNP35N07 70V 0.0280 35A 125W  
 VNP49N04 42V 0.020 49A 125W

## Requirements

Because two of the wake-up call methods of the alarm are the high power devices (the breeze and dawn simulations), a high current capable switching device will be required for controlling them.

The first idea would of course be to use a relay. However there is one big problem with this; both the fan and lamp are to have their intensity varied which will require varying the voltage across them. The easiest and most efficient way of doing this with a PICAXE is to use the built in pwmout (Pulse Width Modulation) command (only supported in the X versions).

However due to the nature of PWM (rapidly switching the output on and off, with intensity controlled by the duty cycle of the pulses), relays are not suitable switching components for use in a PWM system. This is because they are mechanical components and have a high latency (relative to the length of a PWM cycle).

Therefore I will need to use a solid state switching component. I had a look on the Rapid Online website and found the Power MOSFET described on the left. The specifications are more than suitable for the job, and the low threshold drive means that I should be able to connect it directly to the PICAXE without intermediary components.

I have chosen device VNP49N04 because it will afford me maximum choice with regard to lamp power. The order code is 47-0400, price £2.20



## Research - Electronics - Output (Wake-up Call) Components

### Requirements

I have decided that one of the wake-up methods of the alarm will be a dawn simulation.

Again it is crucial to the quality of the clock that the lamp used is of appropriate output power. Too low a brightness could mean the lamp simply doesn't wake the user.

It is hard to gauge how bright a lamp will be based on its power rating because other things affect the brightness; mainly the way it is mounted and thus efficiency (percentage of light that is actually focused where it should be).

Realistically the only limiting factor for how bright the bulb can be is the power consumption and heat output but it is also important that the lamp is not so bright that it dazzles the user! As I said above brightness is hard to gauge so the most realistic approach I can take is to try the 20W bulb, order code 41-1150, and see if it's bright enough; if not I will need to replace it with a more powerful one.

Unfortunately the power consumption of the bulb cannot be reduced by using an energy saving bulb because energy saving bulbs do not function at low voltage and thus full range brightness control cannot be achieved with PWM (pulse width modulation) or otherwise, and those that can function with PWM (LEDs) are not bright enough unless used in an array and even then have the wrong colour balance for a dawn simulation (even more so than tungsten, that is).

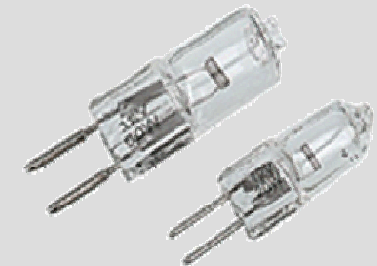
### 12V Tungsten halogen capsule lamps

Price: £0.60.

Found at Rapid Online > Electrical & Power > Elect Prod & Lighting > Lighting > 12V Tungsten halogen capsule lamps

M class 12V halogen capsule bulbs with bi-pin bases often encountered in decorative and low voltage lighting as well as in specialist lighting applications.

- The 20W bulb (type M47) has a G4 base
- The 50W bulb (type M32) has a GY6.35 base



## Research - Electronics - Output (Wake-up Call) Components

### Delta 80mm Fan

An ultra-high performance 80mm case fan pushing a staggering 68.51CFM. Not the quietest fan in the world but certainly one of the best - 68.51CFM, 4900RPM, 48.5dBA. Please note - DO NOT CONNECT THIS FAN DIRECTLY TO YOUR MOTHERBOARD. Its current draw is high enough to damage the motherboard fan headers. Only connect this fan to a compatible thermal control unit such as the Digi-Doc 5 (available from the Overclocking Accessories Section) or directly to your PSU via a 3-pin to 4-pin converter (not supplied).

MODEL: FFBo812SHE

- Speed - 5700RPM
- Output - 80.1CFM
- Decibels - 52.5dBA
- Dimensions - 80x80x38



Price £11.69

Found at Overclockers UK Website: <http://www.overclockers.co.uk/showproduct.php?prodid=FG-004-DE>

### Requirements

I have decided that one of the wake-up methods of the alarm will be a breeze simulation, in addition to the dawn simulation.

The two key factors in deciding which fan is appropriate are size and power/noise (which are closely linked).

This Delta 80mm Fan is, as described opposite, of the powerful and noisy variety. It conforms to the standard dimensions for computer chassis cooling, and also available in the range are 40mm, 60mm, 80mm, 92mm and 120mm diameter versions, which will of course vary in airflow and noise.

Delta specialise in high volume fans with the drawback being the noise; I have chosen to use a Delta fan because it is important that the output is actually noticeable or it serves no purpose!

I think an 80mm fan is the ideal size because any bigger would mean the case would have to be very big, and any smaller would restrict how noticeable the 'breeze' would be.

## Research - Electronics - Output (Control) Components

### Requirements

As described in the task analysis there are two possibilities for the display mechanism; either an LCD or 7-Seg LED displays (Starburst displays would require very complex logic to control so are not realistic).

Having already decided that I will be using a PICAXE microcontroller (due to their ease of use), I had a look for other devices made by Revolution Education because they are simpler to interface than other components.

I found the LCD Module pictured on the left, and it fits the requirements perfectly; not only is the display easily controlled but there is also an optional clock upgrade, which I would otherwise have to research and interface separately.

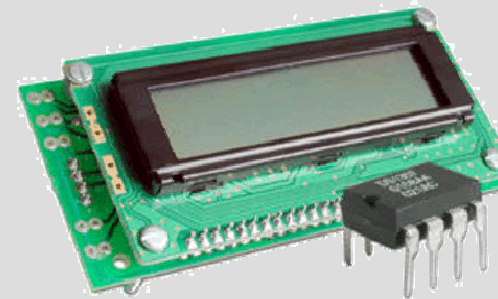
The information shown on the left is all that was written in the product description, however there was a lot more data in the product datasheet which I have included in Appendix A.

### Serial LCD Module

A module that allows microcontrollers systems like the PICAXE or Stamp to display messages on a LCD. Optional clock upgrade (AXE034) adds real time clock and alarm features to the module.

Product number AXE033, found on the PICAXE website at <http://www.rev-ed.co.uk/picaxe/>.

Price: £14.10



## Research - Electronics - Output (Control) Components

### 16 x 2 LCD Screen

Found at Rapid Online > Electronic Components > Optoelectronics > LCDs And Accessories > Alphanumeric LCD display modules

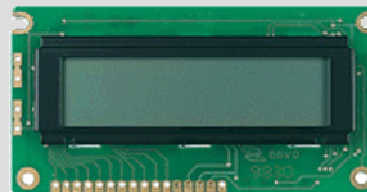
Price: £10.68 (16 x 2 Backlit version), order code 57-0913

A range of intelligent alphanumeric dot matrix display modules employing Supertwist Nematic (STN) technology which provides a superior viewing angle and higher contrast over conventional designs.

Each module uses a 5 x 8 dot matrix format, has a cursor and is capable of displaying 224 different characters and symbols. An on-board RAM facility also enables the user to produce any character pattern required.

The 16 x 2, 16 x 4, 20 x 2 and 20 x 4 modules are available with low power LED backlight to provide excellent contrast characteristics. Intensity of backlighting is also continuously variable over a wide range.

- Very low power consumption (typically 1mA)
- Single power supply +5V
- TTL and CMOS compatible
- Easily interfaced to 4 or 8 bit microprocessors
- CMOS controller and drivers
- Powerful control commands: Display – clear, on/off, shift set function; Cursor – home, address set



### Requirements

The LCD Module I found on the previous page is ideal in every way for the alarm clock except for the fact it doesn't have a backlight.

There are three possible ways around this:

- 1) To not use an LCD at all and instead use 7-Seg LED displays which of course do not need additional illumination (the disadvantage of this is the requirement for driver ICs).
- 2) To use an LCD for some information and 7-Seg LED displays for showing just the present time.
- 3) To use a backlit LCD display.

The problem with using a backlit LCD display is that I can't just use any that I find—it must be 16 x 2 characters and have the same interfacing specification for it to work with the driver board of the Revolution Education module on the previous page.

If I do decide to replace the standard screen with this backlit replacement I will need to do further checking to ensure it is compatible.

## Research - Electronics - Output (Control) Components

### Requirements

As described on the previous page, the lack of backlight on the standard display included with the Revolution Education LCD module is a problem and one of the solutions would be to use 7-Seg displays for showing some or all of the information that must be communicated to the user.

If I were to use 7-Seg displays, these quad module displays would be ideal. The fixed colon version would be used of course, for displaying the time.

I would choose the green version for better visibility (the eyes are most sensitive to green), as well as the fact it is a nicer colour for LEDs in my opinion (too many devices with red LEDs mean that some change is always nice).

For the reasons I already gave in my task analysis, I do not intend to use 7-seg displays.

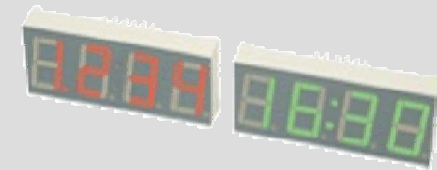
### 4.2mm (0.56) Quad LED Display

Price: Varies from £0.93 to £1.50 depending on version.

Found at Rapid Online > Electronic Components > Optoelectronics > LED Displays > 14.2mm (0.56in) Quad LED display

A range of high quality quad digit LED displays that are available in a variety of colours and with either a floating decimal point or a fixed decimal point/colon. These devices are ideally suited to applications that require readouts of time or varying integers.

- 7 Segment 14.2mm (0.56in) high characters
- Floating point gives readout from 0000 and 0.001 to 9999
- Common anode or common cathode configuration
- White segments with grey display surface maximises on/off contrast
- Connections run along top and bottom of display
- Kingbright 56 series



# Specification

---

I have grouped the specification points based on the aspects of the design that they determine, and ranked them by number based on how important I think they are to the design.

## Wake-Up Call

1. A high power 12V lamp will gradually increase in intensity during the wake-up call over an adjustable period of 0 to 60 minutes that I refer to as the fade-in time. Setting the time period to 0 minutes will cause the lamp to go straight to full intensity.
2. An 80mm diameter 12V DC fan will gradually increase in intensity during the wake-up call over the same adjustable period as the lamp.
3. Once at full intensity, the alarm clock will begin the failsafe wake-up call; a loud piezo sounder. This will continue for a user configured time period that I refer to as the dismiss time.
4. Only one alarm time will be supported to simplify use and development.

## Technical

1. The alarm clock will have an LCD display to show the current time and for feedback during alarm and time setting.
2. It will use the Serial LCD Module AXE033 to provide the display functionality.
3. It will use the AXE034 clock upgrade to provide the clock functionality.
4. A PICAXE will be used as the core of the system (setting time and alarm time, running the wake up call etc.).
5. An X version of the appropriate size PICAXE will be used because I2C support will be needed to communicate with the clock chip and I anticipate that my program will be quite long (the X parts have more program memory and the necessary I2C command support).
6. The alarm clock will be powered by a 12V DC external power supply (batteries aren't sufficient because of the lamp).
7. The correct time will be maintained when external power is disconnected by the backup battery in the AXE034 clock module.

## Control

1. The alarm clock will have no more than five control inputs, to ensure ease of use.
2. All input and output devices will be located on the front face of the case, so that the outputs are effectively directed at the user, and they can easily reach the control panel.
3. To ensure usability in the dark, the control inputs will be illuminated.
4. Rather than having a snooze or dismiss button, the clock should automatically dismiss itself after both gentle and fail-safe wake-up calls have completed. This is to prevent determined users from going back to sleep by dismissing it (unless of course they unplug the power sup-

# Specification

---

ply).

5. There will be no on/off switch because the functionality of an on/off switch is simply not necessary with clocks - they are always on.

## Case

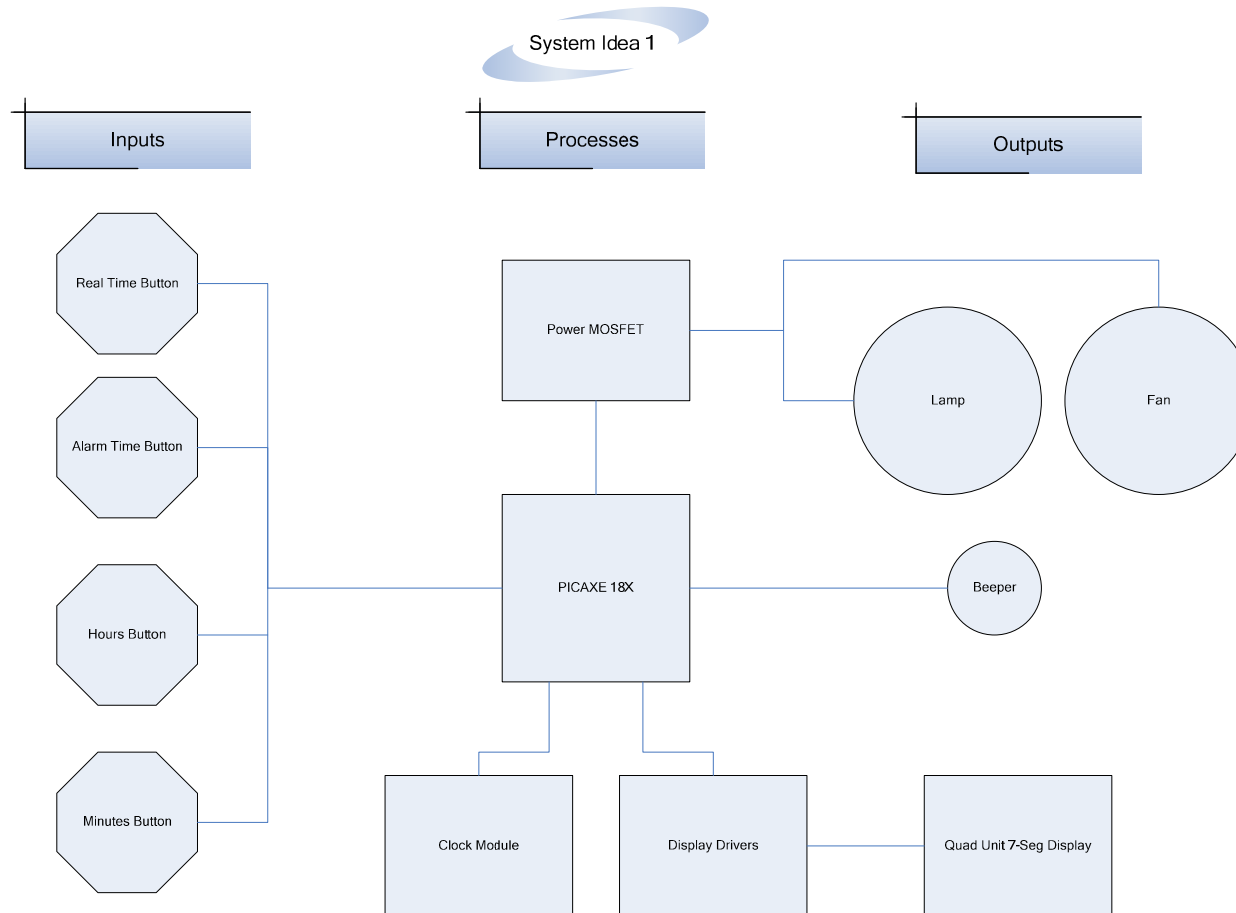
1. The case should be as small as is reasonably possible given the output components I will be using, so that it isn't imposing on the room.
2. All components will be contained within one case—that is, the dawn simulation lamp and breeze fan will be housed within the main case, to ensure convenience when setting up and positioning the alarm clock.
3. Connection to the power supply will be by a DC power jack located at the rear of the case.
4. The circuit board will be firmly attached inside the case using stand-offs.
5. The alarm clock will also double up as a nightlight, so will require a lamp override control on the panel.

## General

1. The project should be completed in less than 40 hours.



# Initial Circuit Ideas - System Diagrams



## Circuit Idea 1

In circuit idea one, the control mechanism is provided by four buttons, as described below.

Pressing the hour or minute buttons while holding down the real time button will increment the real time hour and minutes respectively.

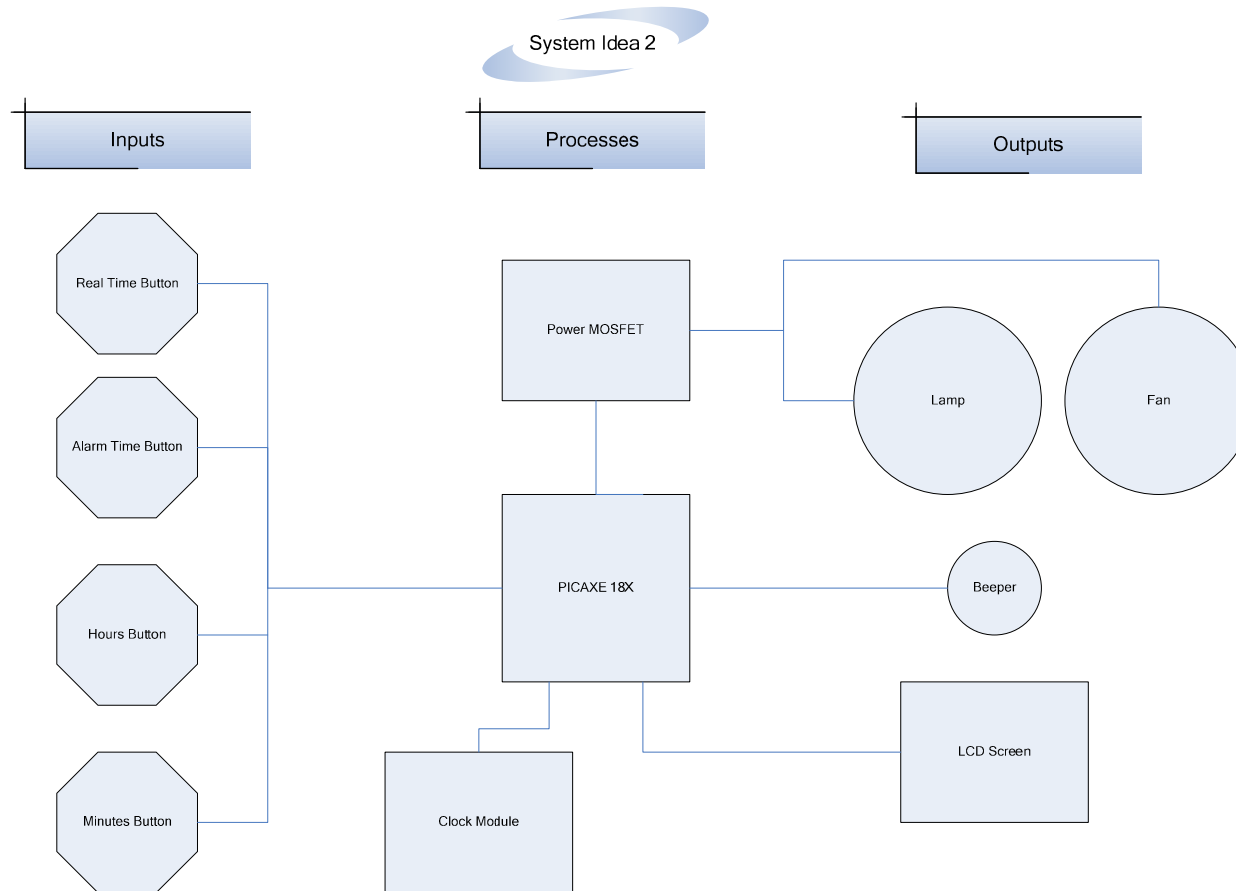
Likewise, pressing them while holding the alarm time button will increment the hours and minutes that the alarm time is set for.

By default the real time is displayed on the quad unit 7-Seg display but while the alarm time button is depressed, the time that the alarm is set for is shown on the display.

This is the only control the user has over the functioning of the alarm clock and as such this idea fails to meet many items on the specification.

The fade-in time is built into the program as is the dismiss-time (see specification).

# Initial Circuit Ideas - System Diagrams



## Circuit Idea 2

In circuit idea two, the control mechanism is the same as that in idea one. The difference is that idea two uses a two line LCD screen to display information.

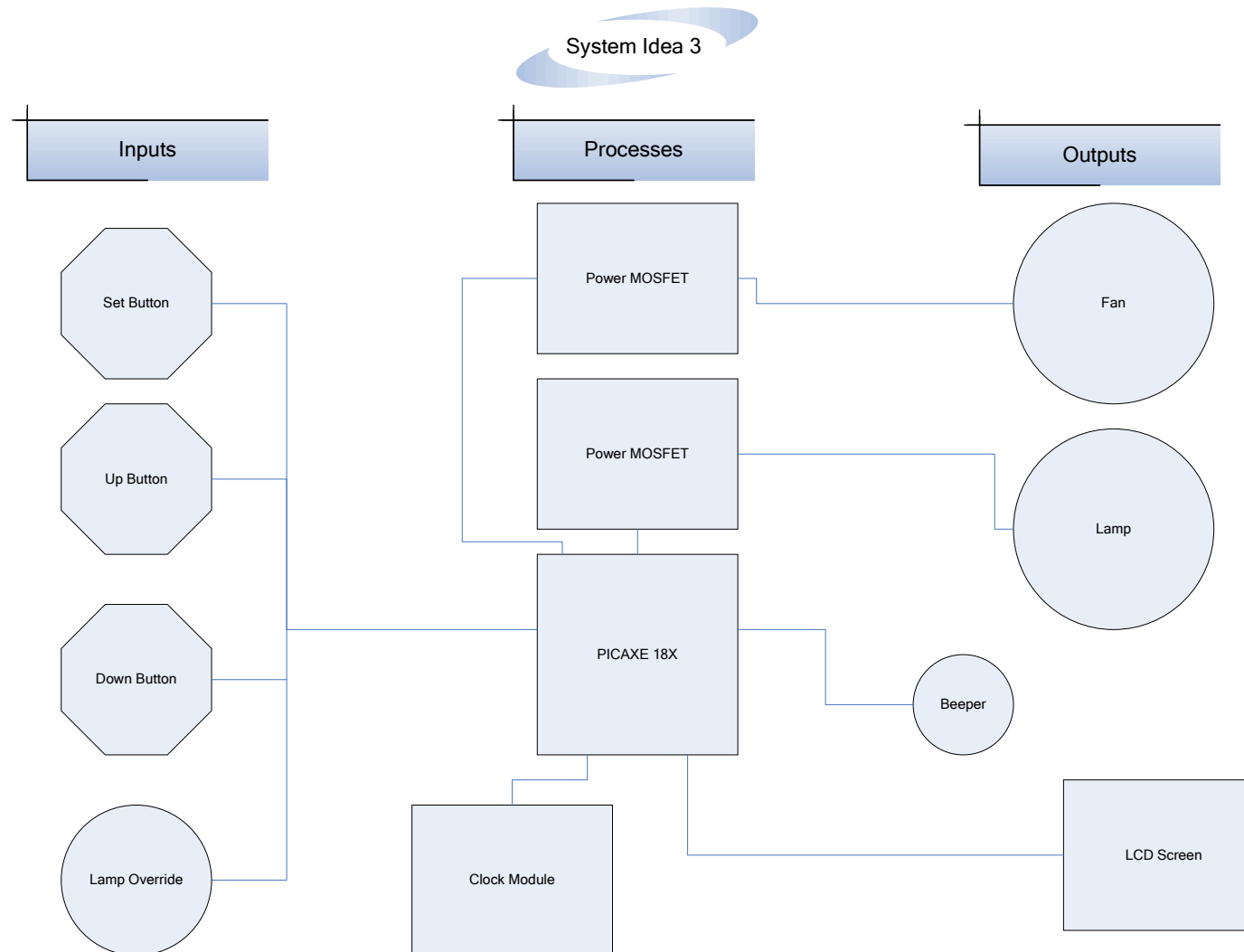
The upper line states what information is being shown, and the lower line actually shows the information.

By default the screen would show 'Time:' on the top line and '19:57' (for example) on the bottom line.

If the alarm time button is pressed then the LCD would read 'Alarm time:' on the top line and '6:00' (for example) on the bottom line.

The only real advantage of this idea over idea 1 is that using an LCD should simplify programming, make the control mechanism ever so slightly easier to use (due to the screen stating what the current view is).

# Initial Circuit Ideas - System Diagrams



# Initial Circuit Ideas - Idea Three Program Description

---

## Circuit Idea 3 (Chosen idea for development)

In circuit idea three, the changes are:

- a) The control methods
- b) The control provided

The change to the control methods is that the alarm time, real time, hour and minute buttons are replaced with a set, up and down button, and a variable resistor labelled lamp override. By default, 'Time:' is shown on the upper line of the LCD and '19:57' on the lower time (if the time was 19:57). If the set button is pressed, the menu system is initiated and the first menu page—the time setup—is displayed.

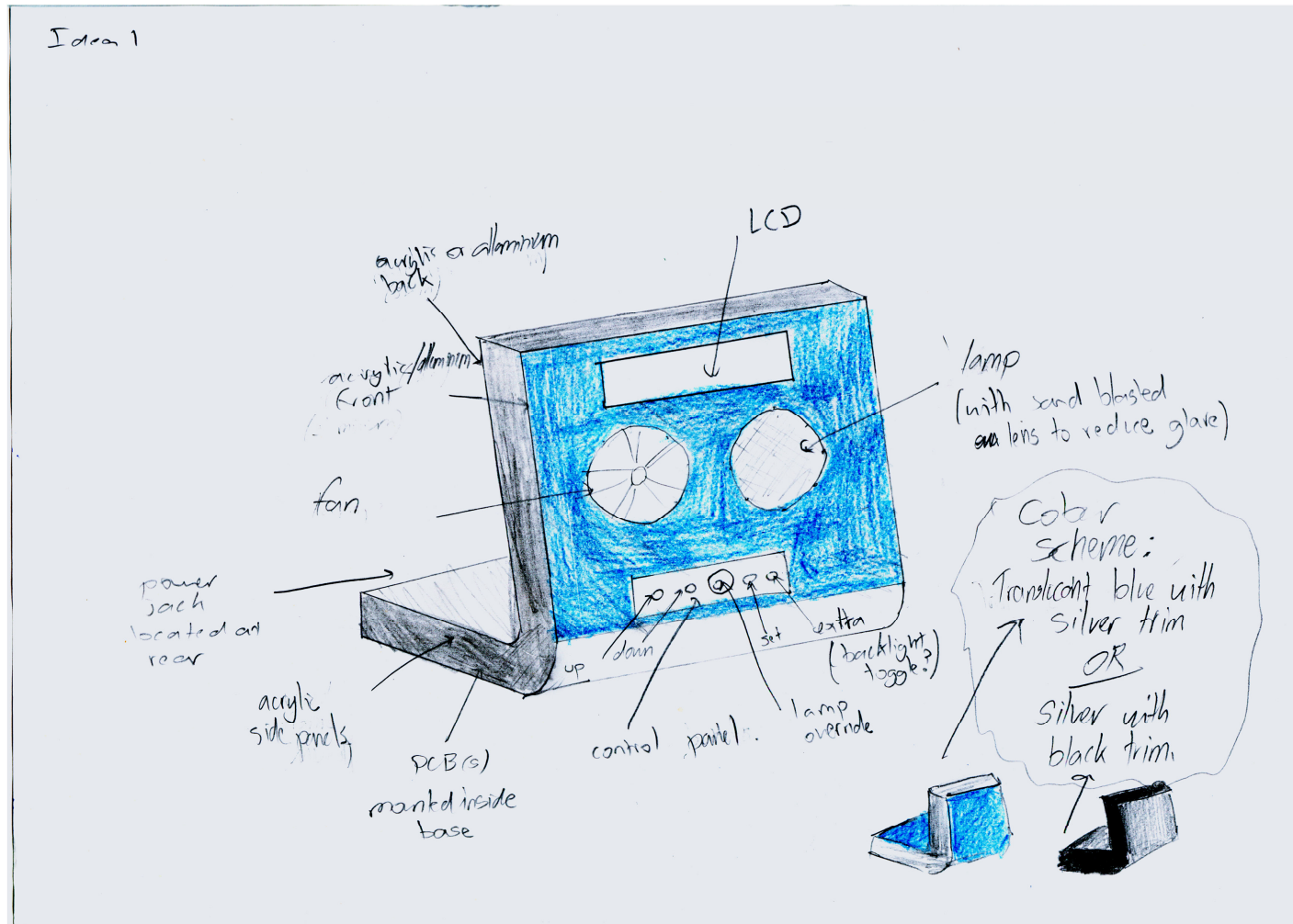
Once in a setup screen, pressing the up button will increase the selected value and pressing the down button will decrease it. Pressing the set button will move the cursor to the next value on the setup screen, and when the last value is reached on that screen, the next screen is shown. When all the parameters and screens have been cycled through the menu closes and the alarm shows the real time. This will also happen if there is no user input for 15 seconds (to prevent accidental changes if the alarm clock is left and buttons pressed unintentionally).

There is an additional control, labelled 'Lamp Override'. This is a rotary potentiometer, as found on domestic lamp dimmers. When in the off position, the alarm functions as normal. When rotated however, the alarm clock doubles up as a dimmable bedside lamp. This is the reason for having two power MOSFETs in the circuit; to allow individual control over the lamp and fan. However one disadvantage of this is that support for the pwmout command on at least two pins will be required from the PICAXE. An alternative approach to this would be to have only one pwm output and then switch the fan on/off by means of an additional MOSFET (normally on but off when the alarm clock is used as a lamp).

The user configurable parameters are: the real time, the alarm time, the fade-in time, and the auto dismiss time. The alarm clock will not have a dismiss or snooze button for reasons described in the project outline and task analysis (to remove the option for the user to just go back to sleep—it is for heavy sleepers!). There will be a separate menu screen for each user configurable parameter, and they will cycle in the order I have listed them above.

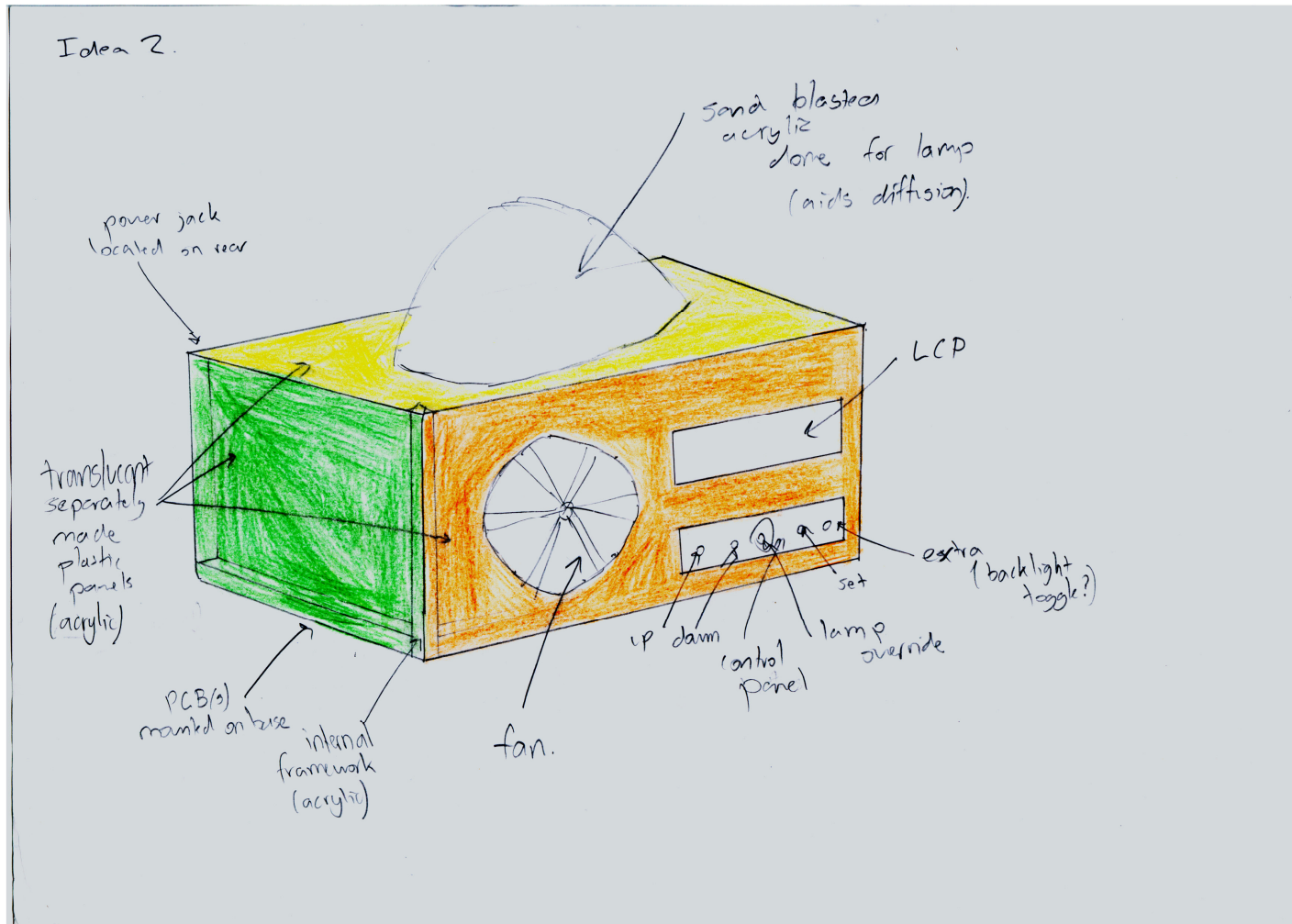
# Initial Casing Ideas

## Case Idea One



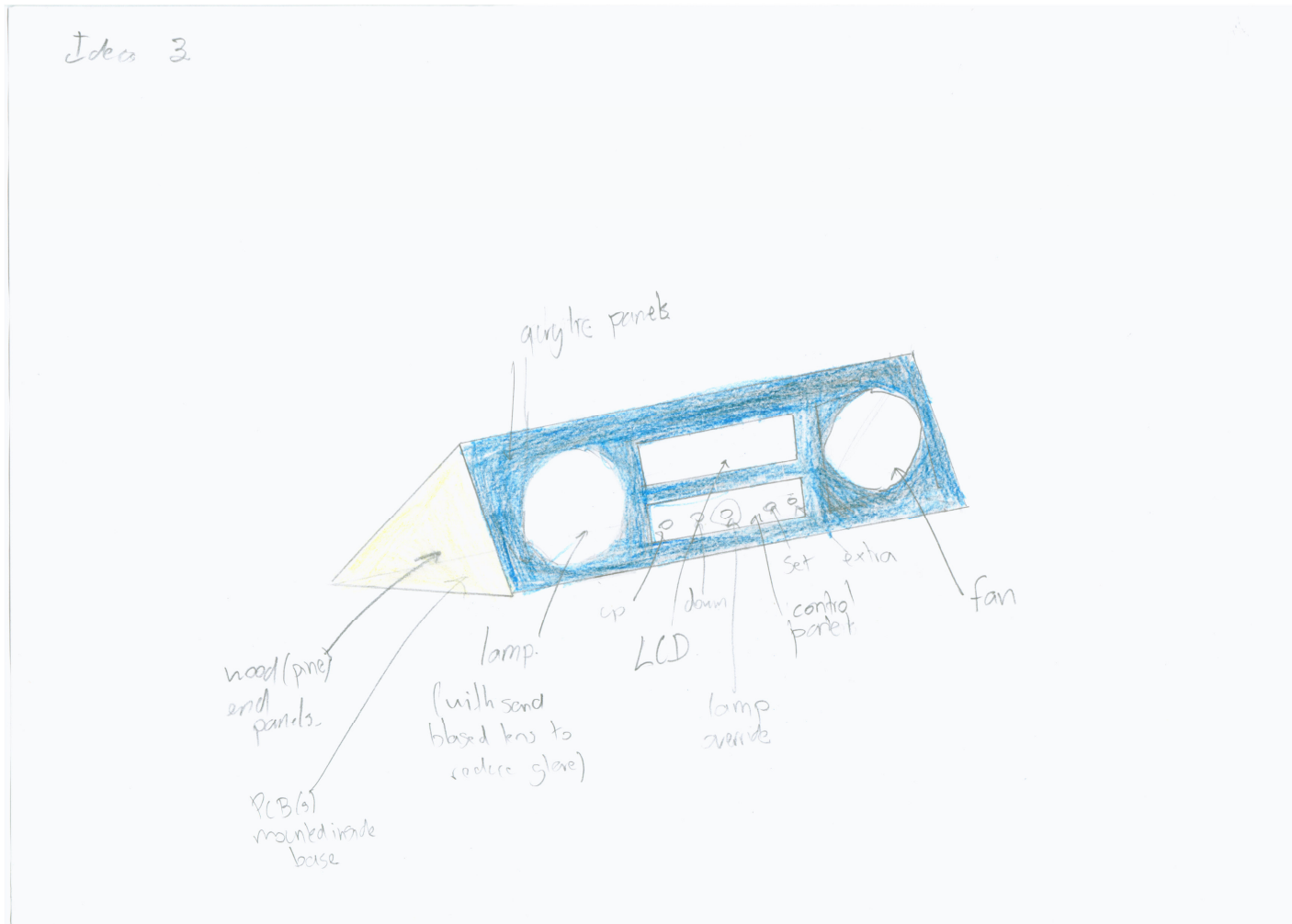
# Initial Casing Ideas

## Case Idea Two



# Initial Casing Ideas

## Case Idea Three





## Development - Breadboarding & Programming

---

### AXE033 LCD and AXE034 Clock Upgrade

The AXE033 is a 16 x 2 LCD with a serial driver board attached on the rear, with support on the driver board for a Dallas DS1307 clock IC and backup battery for when power is removed. This is an optional upgrade. It is supplied in a partly assembled kit; the two boards were populated but I had to solder and attach them together, and install the clock upgrade kit.

When the AXE034 clock upgrade is installed the unit also has support for alarms at set times. The driver board supports two communication systems; I2C and serout (which is used is controlled by a jumper switch on the driver board).

### Serout advantages & disadvantages

The key benefit of connecting the driver board to the PICAXE in the serial mode is the simplicity; only one data wire is needed and the serout command is supported across the whole PICAXE range. Serout is the generally recommended setup in the datasheet (see appendix A).

In this mode, all control of the DS1307 clock IC is by proxy of the serial driver IC (that is, the entire kit can be treated as a single unit when it comes to programming). The advantage of this is that it simplifies the displaying of the time on the LCD which can be achieved with the command:

```
serout 0,N2400,(0)
```

where 0 is the number of the PICAXE pin that the serial wire is connected to and the N2400 specifies the baud rate of the connection.

However, serial communication in this system is monodirectional; that is, the PICAXE is able to send data to the LCD driver board but not the other way round. Bidirectional communication is required so that the real time and alarm time can be read from the clock module. This is so that when the user enters the real time or alarm time setting screens, the string that they edit initialises with the values of the current setting.

Perhaps this is better described by saying that without this functionality, a user could set the alarm time for 6am, leave the setup menus, return to the setup menus and find that the alarm clock reports the alarm is set for 00:00:00 (because the PICAXE doesn't know otherwise),

## Development - Breadboarding & Programming

---

when in fact it is still set for 6am. This is far from user-friendly!

### I2C advantages & disadvantages

The major disadvantage of using I2C is that I2C is only supported on the PICAXE X parts which are more expensive. An additional disadvantage of using I2C is that two data wires are required (clock and data), though I do not consider this to be significant.

The final point is one that can be seen as a good and bad thing; the kit can no longer be treated as a single unit. When using I2C (see appendix A for datasheet with information on the I2C protocol), the LCD driver IC and DS1307 are two separate I2C slave devices. Communication with the DS1307 is no longer by proxy of the LCD driver IC. This means that:

1. Communication is now bidirectional meaning that data can be read from the DS1307. This is very important as it will enable me to write the program such that the values of settings are printed back to the screen as they are edited. This feedback is crucial to the usability of the product and so for this reason, communicating with the DS1307 (and thus using I2C) is a must.

A further example to clarify this, would be if the user is changing the value of the fade-in time. In order to change this setting to 15 minutes, for example, they must be able to see what value it was at previously so that they know how many times to press the up/down button.

2. The disadvantage of using I2c to communicate with the LCD driver IC and DS1307 is that because the DS1307 is now accessed directly, the ability to print the time with the simple command:

```
serout 0,N2400,(0)
```

3. is no longer available.

### Conclusion

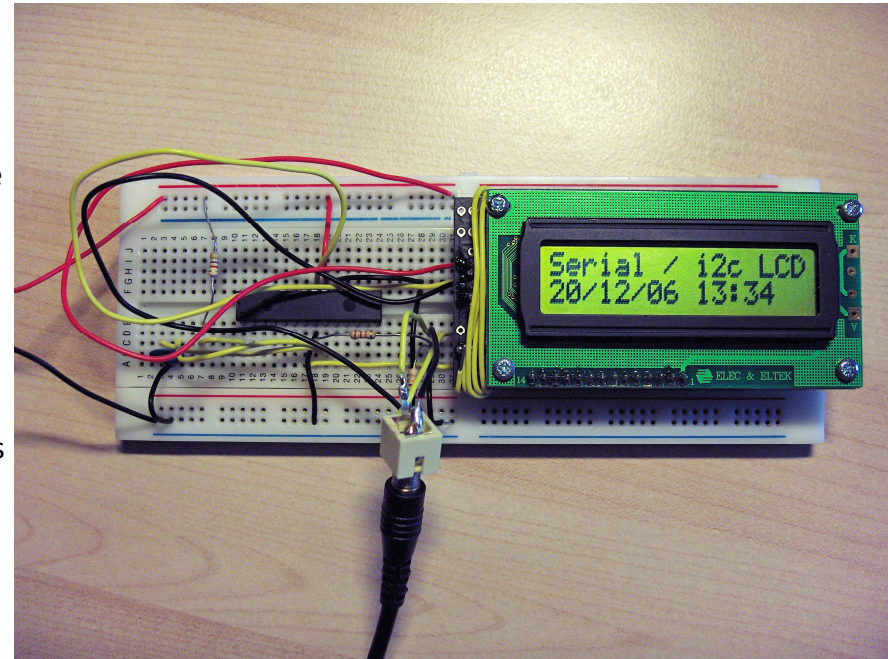
In conclusion it is crucial to the usability of the alarm clock that I am able to communicate directly with the DS1307 and therefore that I use the I2C protocol for communicating with the LCD driver IC and DS1307. Since the PICAXE X parts are the only ones to support I2C, it is necessary that I use a PICAXE 18X (the smallest version with sufficient IO, features and memory).

## Development - Breadboarding & Programming

### Programming

I have decided to use BASIC and the free Programming Editor software provided by Revolution Education. This is for three reasons:

1. Text based programming is my personal preference - I do not believe graphical methods are flexible enough;
2. I anticipate that my program will be very long and complex compared to the sort of thing Logicator 2005 was designed for, and thus would be more complicated on Logicator 2005 than in BASIC.
3. Because I will use the I2C protocol for communication between the PICAXE, LCD driver and DS1307, the development system I use will of course need to support the I2C command set; I attempted to use this in PIC Logicator 2005 but it doesn't support it. So instead, I used the facility it has that allows the insertion of BASIC code, inserting all the lines that would need the I2C commands in this way. However, on clicking the button to program the PICAXE, I received an error telling me it didn't recognise the I2C commands. Thus, even if I had wanted to use PIC Logicator 2005, I wouldn't have been able to.



Picture of the PICAXE 28A, AXE033 (LCD) and AXE034 (clock upgrade) connected serially.

### Familiarisation

Prior to having completed a lot of the research, analysis and development that precedes this section I was given a PICAXE 28A, AXE033 and AXE034 upgrade kit to try out. Although for reasons explained on the previous page the 28A is not suitable (no I2C support as it is an A, not X version), I took the opportunity to familiarise myself with the AXE033 and 034 kits using the serial protocol.

After having assembled the kit I constructed a basic circuit on breadboard that would enable me to control the LCD via the PICAXE 28A. This can be seen top right. I have not included a schematic because the circuit is pretty much identical to the example circuits in the datasheets.

## Development - Breadboarding & Programming

---

The first program I wrote was one to set the correct time and date (shown on screen in the picture on the previous page). This was fairly straightforward:

```
1  init:
2      pause 500
3  main:
4      serout 0,N2400, (253,0,"20/12/06 13:34 ")
5      end
```

Line 1 is what is called a label; it defines the start of a point in the program that can then be referenced in control structures (if statements, goto commands etc.). When not used for this they can be useful simply for the purpose of readability.

Line 2 makes the program pause for 500ms. This is recommended in the AXE033 datasheet to allow the LCD driver IC to initialise (hence that section is labelled init).

Line 3, another label, defines another section of code - the 'main' part of the program. Line 4 is the actual serout command that sends on pin 0 the number 253, followed by the number 0, followed by the ASCII code for each individual character in the double quotes (see appendix A for ASCII conversion table). Line 5 instructs the PICAXE to stop program execution.

To check that this was successful I closed the CLK jumper on the LCD driver board, which as stated in the datasheet causes the LCD to run in a sort of standalone mode, independent of (and not requiring) that the PICAXE is connected to it. This is a useful mode for testing the LCD and DS1307. This was successful and the screen output was that shown in the picture on the previous page.

Happy that the serial communication was working correctly, I set about writing another test program to further help familiarise myself with controlling the LCD module.

## Development - Breadboarding & Programming

---

I wrote this program solely to gain some experience with BASIC. I have used labels excessively (most are not strictly necessary as they aren't used in program flow control structures) because they make the different stages of the cycle clearer. I have also been generous with my commenting of the program (comments are as the name suggests, comments left in the source code of a program to help in understanding/remembering how it works). In BASIC the start of a comment is signified by a single apostrophe and continues until the end of the line. Comments are ignored by the compiler which means they have no effect on the functioning of the program, and do not alter the amount of memory needed in the PICAXE. The program is non-interactive and the sequence is as follows:

1. It initialises with a 500ms pause, allowing the LCD to initialise.
2. After this, the classic "Hello world!" programming smoke test is printed to the LCD.
3. One second later the program prints on the top line "The time is:" and shows the time on the bottom line (because serout is being used this is as simple as sending the number 0 to the driver IC which instructs it to print the time).
4. Two seconds after this, the program fakes a "crash" by outputting seemingly random characters to the screen. This is achieved with a loop that sends each number between 150 and 240 in sequence to the screen. The key to how and why this works is in the fact that I have not placed the variable reference `bo` in quote marks - if I had, the text `bo` in the source code would be interpreted as a constant and thus 'bo' would be printed to screen. The second important thing to realise is that because I have not specified otherwise in the program, the value of `bo` is not interpreted by the PICAXE but instead sent directly to the LCD driver IC. Because I selected an appropriate range for the loop (150 to 240), the driver IC interprets the numbers it receives as ASCII characters and thus prints what are seemingly random characters. As stated in the LCD datasheet, the driver automatically wraps onto the next line after 20 characters which is why I have not included a command to move the cursor to the second line.
5. After some time the program comes out of its pretend crash and simulates a reset, by announcing "FATAL ERROR" on the top line and that it is "RESETTING" on the bottom line. This is followed by ellipsis that are printed one by one to the screen, with half a second between each one. As with the earlier crash simulation, this is done with a "for... next" loop. This time the variable `bo` is used in a control command that changes the location of the cursor on screen (with an appropriately selected range of numbers - 192 is the start of line 2 and so 201 is 9 characters from the left. Once the ellipsis reach the end of the screen, the program loops to "hello" and runs again.

At the end of the program there is a subroutine, "clear", called to clear the display. This saves program memory as the code is only included once, and any changes only have to be made once (not really that useful for such a small routine but it will be later) in the real program.

## Development - Breadboarding & Programming

---

```

1  init:          28          ' Output random ASCII char- 49          pause 1000 ' More effect time
2      pause 500  29          50
3      30          51  blink:
4  hello:        31          for b0 = 150 to 240 ' de- 52          ' Print a series of dots
5      ' Clear display 32          fine loop for ASCII Set 53
6      gsub clear  33          serout 0,N2400,(b0) 54          for b0 = 201 to 207 ' define
7      34          pause 100 55          loop for 6 times
8      ' Be polite! 35          next b0 ' end of loop 56          serout 0,N2400,(254,b0)
9      serout 0,N2400,("Hello 36          57          serout 0,N2400,(".")
world!") 37          58          pause 500
10     38          59          next b0 ' end of loop
11     ' Give them time to read it 39          60          ' Done!
12     pause 1000 40          ' Blink - clear display 61          goto hello
13     41          62
14  thetime:    42          ' A bit of time for effect 63  clear:
15     ' Clear display 43          pause 1000 64          ' Clear display
16     gsub clear  44          65          serout 0,N2400,(254,1)
17     45          66
18     ' Be patronising 46          or not 67          ' Give it some time...
19     serout 0,N2400,("The time 47          serout 0,N2400,(254,128) ' 68          (required)
is:") 48          Move to top left 69          pause 30
20     49          70          ' Move cursor to top left
21     ' Print the time 50          serout 0,N2400,("FATAL ER- 71          serout 0,N2400,(254,128)
22     serout 0,N2400,(0) 51          ROR.") 72
23     52          pause 500 ' That crucial 73          ' Done!
24     ' Give them time to read it 53          "effect time" again 74          return
25     pause 2000 54          serout 0,N2400,(254,192) '
26     55          Move to top right
27  crash:      56          48          serout 0,N2400,
("RESETTING")

```



## Development - Breadboarding & Programming

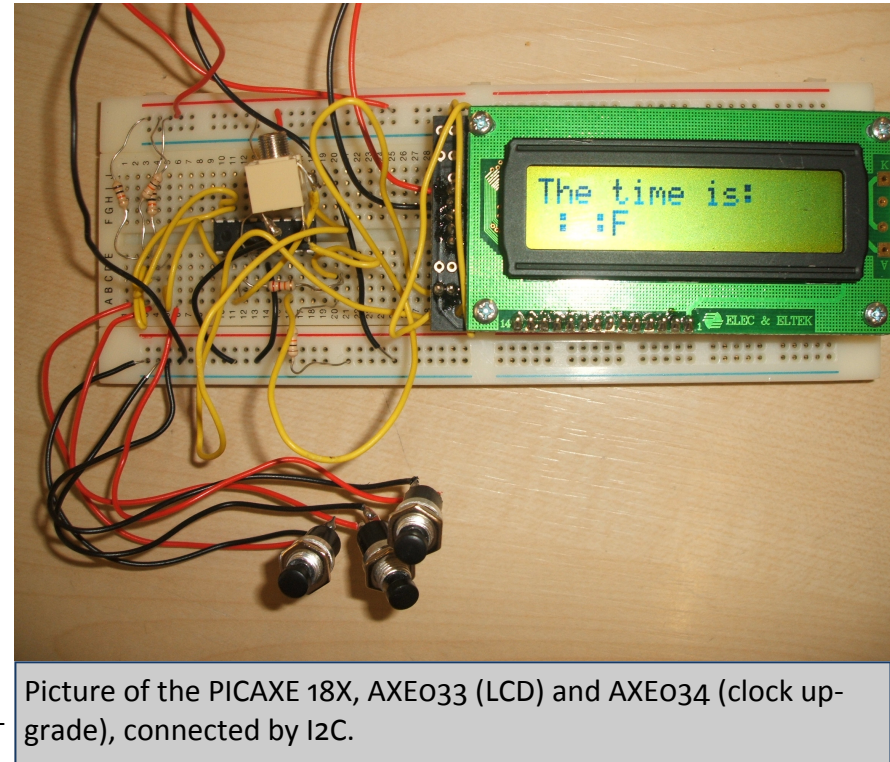
### Breadboard Changes

The test program just described was successful and worked as intended. I now had available to me a PICAXE 18X meaning that I could change the breadboard and have the PICAXE connect to the LCD unit by I2C instead of serout. Connection was relatively simple and just required soldering the clock and data wires to the LCD unit and connecting them to the clock and data pins of the 18X on the breadboard (pins 10 and 7 respectively). The updated breadboard is pictured top left.

I also decided to add three inputs, ready for when I start programming the setup menu system. I used 10K pull up resistors to create a potential divider for each. The switches are normally closed and so I connected them such that they would normally be holding the PICAXE inputs low.

I now felt ready to start working towards the real program. Rather than trying to write the whole thing in one go I decided to add small parts at a time. The reason for doing this is because it is easier to debug program elements on their own as opposed to when they are part of the whole program.

The first part I wrote is the part that writes the time to the LCD and continually updates it. By having done this I am now aware of a major problem I would otherwise have encountered later on in development. The program is included overleaf.





## Development - Breadboarding & Programming

---

```

1  init:
2      ' Let the LCD init
3      pause 500
4
5      ' Set up the i2c bus
6      i2cslave
7      $C6,i2cslow,i2cbyte
8      ' Set up the input symbols
9      symbol UP = pin0
10     symbol DOWN = pin1
11     symbol SET = pin2
12
13     ' Set up time variables
14     symbol seconds = b3
15     'seconds
16     symbol minutes = b4
17     'minutes
18     symbol hours = b5 'hours
19
20 main:
21     ' Clear display
22     gosub clear
23     ' Read the current time
24     from the clock chip
25     i2cslave %11010000, i2c-
26     slow, i2cbyte ' set slave details
27     readi2c 0, (seconds, min-
28     utes, hours) ? read sec, min, hour
29     i2cslave
30     $C6,i2cslow,i2cbyte ' reenable LCD
31     control
32     ' Print the current time
33     writei2c 0,("The time
34     is:",254,192,hours,":",minutes,":",s
35     econds,255)
36     pause 1000
37
38     ' Loop
39     goto main
40
41 golower:
42     ' Moves cursor to lower
43     left
44     writei2c 0,(254,192,255)
45
46     ' Done!
47     return
48
49 clear:
50     ' Clear display
51     writei2c 0,(254,1,255)
52
53     ' Give it some time...
54     (required)
55     pause 30
56
57     ' Move cursor to top left
58     writei2c 0,(254,128,255)
59
60     ' Done!
61     return

```

## Development - Breadboarding & Programming

The schematic of this new circuit is shown top right.

### I2C Control Differences

As with the other programs this one features the 500ms init time. However the next line is new; when using I2C, you must specify the address of the slave device that you wish to communicate with prior to any readi2c/writei2c commands. This is explained in greater detail in the datasheet about I2C published by Revolution Education, which I have included in appendix A. I do this on line 6 with this command:

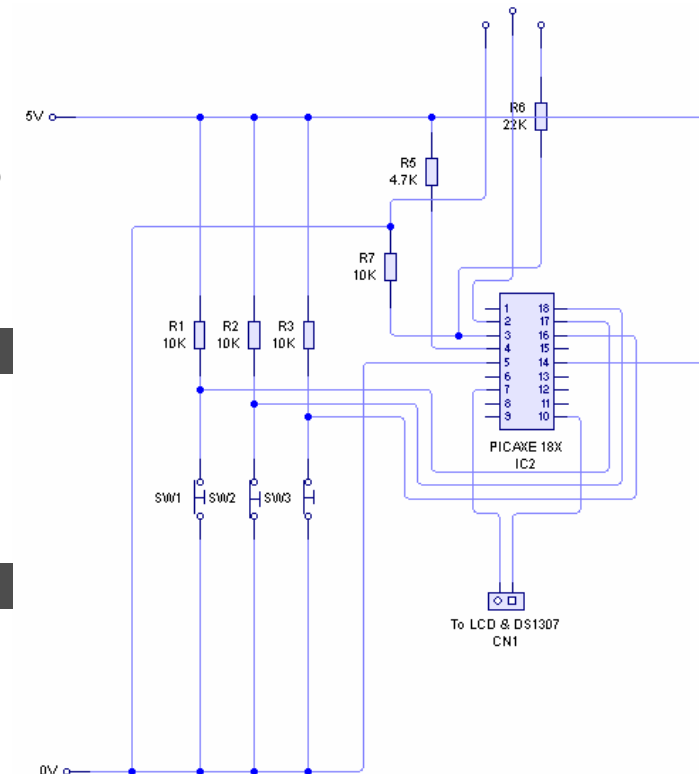
```
i2cslave $C6,i2cslow,i2cbyte
```

The '\$C6' parameter is the actual address (in this case the LCD driver ICs), while i2cslow and i2cbyte specify the speed of communication and register address size. Read/write commands take identical parameters except for the name of the command itself. Example read command:

```
writei2c 0,(254,1,255)
```

The first parameter, the 0, specifies the starting address to read/write from (and should normally be written as a BCD - binary coded decimal - it is because I am specifying address 0 that the notation makes no difference). The PICAXE will then continue reading/writing, advancing one data unit (byte or word depending on the register address size) each time until it reaches the end of the comma separated values contained/referenced within the brackets.

In the case of controlling the LCD, each value is interpreted as a command and together they form a command sequence. The actual control numbers for the LCD are the same as those when using serout, except that the command sequence must be terminated with the number 255 which tells the LCD driver IC that the command sequence is complete. It should be obvious therefore, that the above command will clear the screen (as 254 sets the driver to control mode, 1 tells it to clear the display, and 255 tells it the command sequence is complete).



Schematic of the new circuit (LCD and DS1307 not shown - they were connected at CN1).

# Development - Breadboarding & Programming

---

## Program Explanation

Also new in this program is my use of symbols which are simply a means of improving the readability of the program. Although I didn't actually use the newly added push to break switches in this program, I defined symbols for them so that if I did, rather than having to look which pin each was connected to every time I wanted to use it in the program, I could have just referred to it by its symbol. I also defined symbols for some of the byte variables. Again this improves readability and makes modifying the program easier, because rather than having to remember what data I had loaded into a variable, I know what data is loaded into it by its name. This of course only works provided that I do not load data that doesn't match the descriptive name into the variable.

The last addition unaccounted for is the `goupper` and `golower` subroutines. I wrote these ready for use later in programming, and as with the push to break switches didn't actually use them in this program.

The general flow of the program can be described as follows:

1. Init; I2C setup (for LCD communication) and symbol definitions.
2. Clear display.
3. Setup I2C for communication with DS1307 and read the seconds, minutes and hours from it into the appropriate variables.
4. Setup I2C for LCD communication. Write out the time to the LCD.
5. Pause for one second.
6. Loop back to start (number 2 above - it is not necessary to run the init block each time).

Numbers 1 and 2 have already been covered. It should be reasonably obvious to recognise the section which achieves number 3 in the program. Prior to reading from the DS1307 however, I had to first of all include another `i2cslave` command to setup the PICAXE for communication with the DS1307. The hours, minutes and seconds are then read from the DS1307 into the hours, minutes and seconds variables (as defined by the labels I created in the init block).

Numbers 5 and 6 are so simple they need no explanation. Number 4 however is yet more complex than it appears in the program and lead to a lot of difficulty. See overleaf for explanation.

## Development - Breadboarding & Programming

---

### Writing the time to the LCD

As can be seen in the picture on page 34, the program did not function as desired - it reads " : :F". I quickly realised that what was happening was that the values read from the clock chip into the byte variables of the PICAXE were being sent unprocessed to the LCD. The command used to print the time to the LCD is as follows:

```
writei2c 0,("The time is:",254,192,hours,":",minutes,":",seconds,255)
```

When sending data to the LCD driver IC, it assumes that all data sent is to be printed to the LCD from the current cursor location unless otherwise specified (by preceding the control command with the number 254 as already explained). Additionally, when sending literal text to be printed it is necessary to enclose it in quotation marks. The reason for this is that the PICAXE must be told to convert the characters to their ASCII numbers and this achieved with the quote marks. It should also be obvious that referencing the variables within the quote marks would not work, as this would just result in the following being literally printed:

```
The time is:  
hours:minutes:seconds
```

Therefore it is necessary to reference all variables outside of quote marks. However there is one slight problem with this; as I already said, information not enclosed in quote marks is sent directly without the PICAXE performing any conversion to ASCII. I exploited this in the earlier test program I wrote that included a fake "crash" as part of its operation as it allowed me to loop through a range of numbers and send them to the LCD; the end result being that not the numbers, but their corresponding ASCII characters were printed.

Since I had already exploited this in creating the fake "crash", I should perhaps have realised before writing this program what the output would be. The resulting screen output when the program was run can be seen in the picture on page 34 - it reads " : :F". To clarify, this occurred because the values in the hours, minutes and seconds variables are sent directly to the LCD; no ASCII conversion is performed on them. The reason that the hours and minutes appear blank is because their values correspond to non-printable characters (most characters up to character 32 are non-printable, and since the hour will never be greater than 24 and it was during the first half of the hour that I tested it, the hours and minutes appeared blank.

# Development - Breadboarding & Programming

---

## Converting strings to ASCII character numbers

The following is an extract from the AXE033 and AXE034 datasheet published by Revolution Education and included in appendix A:

Note that the # symbol tells the microcontroller to output the ASCII equivalent of the variable value, not the direct value (e.g. "6" "5" not the value 65, which would actually appear as the character "A"!)

in relation to a line in a sample program also in the datasheet:

```
serout 7,N2400,(254,137,#b1," ")
```

What this means is that it should be possible to make the PICAXE look up the corresponding ASCII number for the values contained within the hours, minutes and seconds variables of my program and hence the program should work. This would be as simple as changing line 28, from

```
writei2c 0,("The time is:",254,192,hours,":",minutes,":",seconds,255)
```

to

```
writei2c 0,("The time is:",254,192,#hours,":",#minutes,":",#seconds,255)
```

However things were not to be and this was just the beginning of the problem.

# Development - Breadboarding & Programming

## Compilation Error

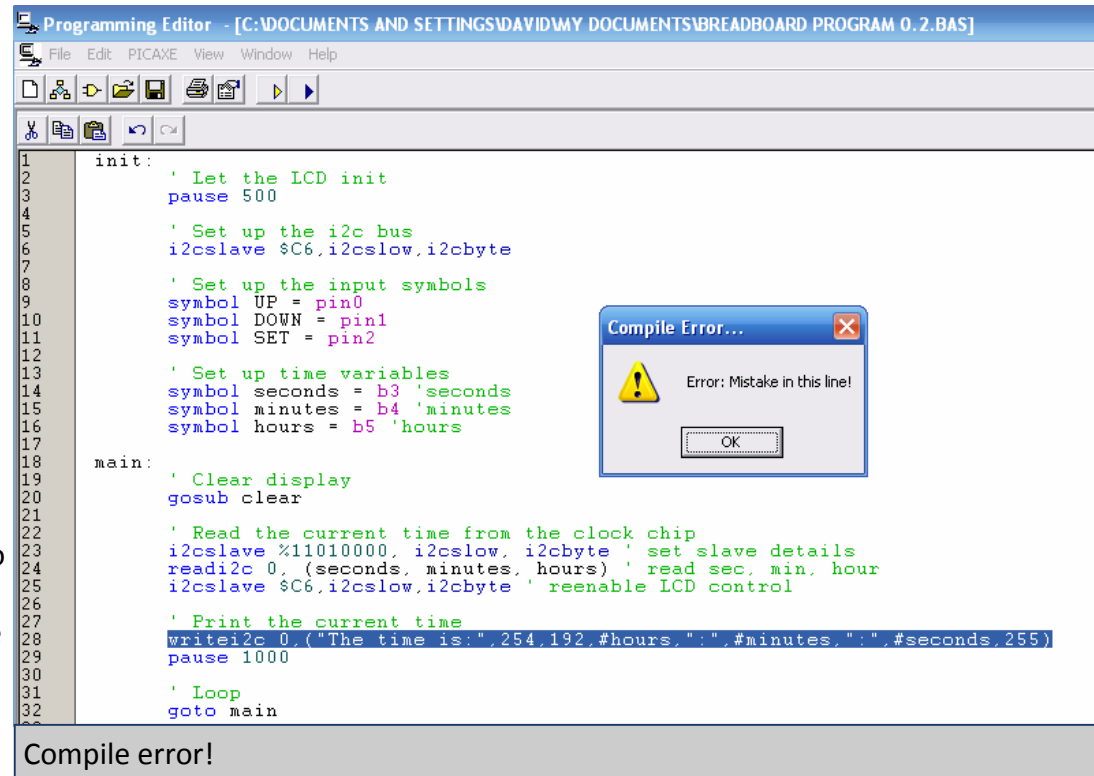
When attempting to program the PICAXE with the amended program, I received a compilation error like the one shown in the screenshot, top right.

I checked every datasheet published by Revolution Education, and the user forum at <http://www.rev-ed.co.uk/picaxe/forum/> but did not find anything regarding the printing of variables with I2C and so concluded ASCII conversion is only possible when using serout, not I2C.

## Paradox: I2C required, Serout required

For reasons already described, I have to use I2C in order to communicate with the DS1307. Yet for the reason described above, it seems I am going to have to interface the PICAXE with the LCD driver IC by serout.

The problem with this is that the AXE033 and AXE034 kits simply do not support this; you can either use serout or I2C but not both.



Programming Editor - [C:\DOCUMENTS AND SETTINGS\DAVID\MY DOCUMENTS\BREADBOARD PROGRAM 0.2.BAS]

File Edit PICAXE View Window Help

```

1 init:
2   ' Let the LCD init
3   pause 500
4
5   ' Set up the i2c bus
6   i2cslave %C6,i2cslow,i2cbyte
7
8   ' Set up the input symbols
9   symbol UP = pin0
10  symbol DOWN = pin1
11  symbol SET = pin2
12
13  ' Set up time variables
14  symbol seconds = b3 'seconds
15  symbol minutes = b4 'minutes
16  symbol hours = b5 'hours
17
18 main:
19  ' Clear display
20  gosub clear
21
22  ' Read the current time from the clock chip
23  i2cslave %11010000, i2cslow, i2cbyte ' set slave details
24  readi2c 0, (seconds, minutes, hours) ' read sec, min, hour
25  i2cslave %C6,i2cslow,i2cbyte ' reenable LCD control
26
27  ' Print the current time
28  writei2c 0, ("The time is:",254,192,#hours,".",#minutes,".",#seconds,255)
29  pause 1000
30
31  ' Loop
32  goto main

```

Compile Error...  
Error: Mistake in this line!  
OK

Compile error!



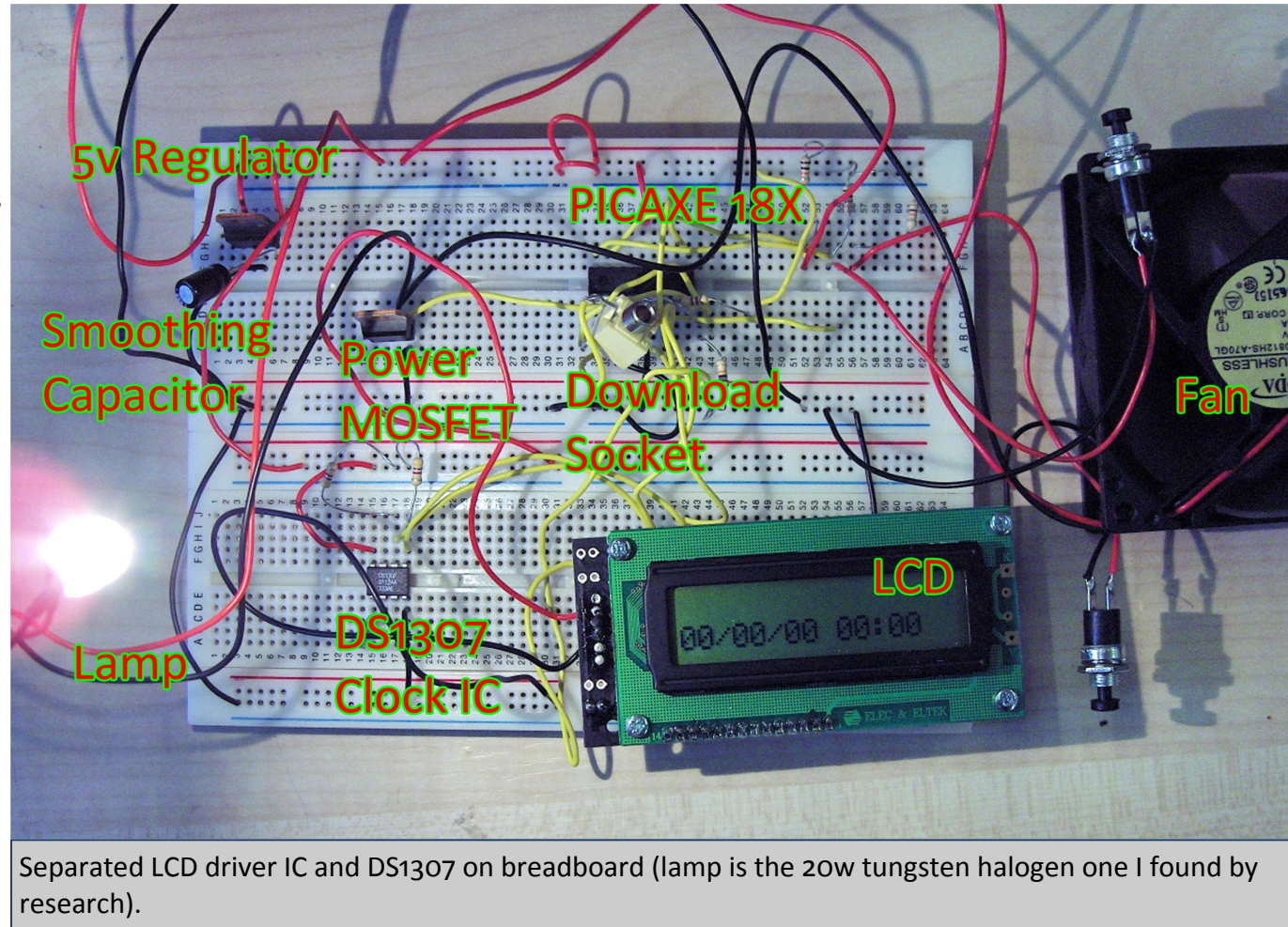
## Development - Breadboarding & Programming

It is essential to the project that the DS1307 is interfaced directly, and essential that the LCD driver IC is commanded by serial out. The only solution I could come up with to this was to separate the LCD driver IC and DS1307, by connecting up the DS1307 on breadboard with the PICAXE, and connecting the PICAXE to the LCD driver IC by serial out.

Doing so required that I knew the pin layout of the DS1307. I found that all the information I needed was in the I2C guide published by Revolution Education.

Connecting it all up on breadboard was relatively simple, though I needed a 32.768 kHz oscillator for it. Rather than trying to source a new one it was quicker to desolder the one on the LCD module, so that is what I did.

The breadboard that I setup to test all this is shown bottom right. The fan, lamp and a power MOSFET (controlling both) are also connected because I had started work on the wake-up functionality of the system, though this will be documented later on.



Separated LCD driver IC and DS1307 on breadboard (lamp is the 20w tungsten halogen one I found by research).



## Development - Breadboarding & Programming

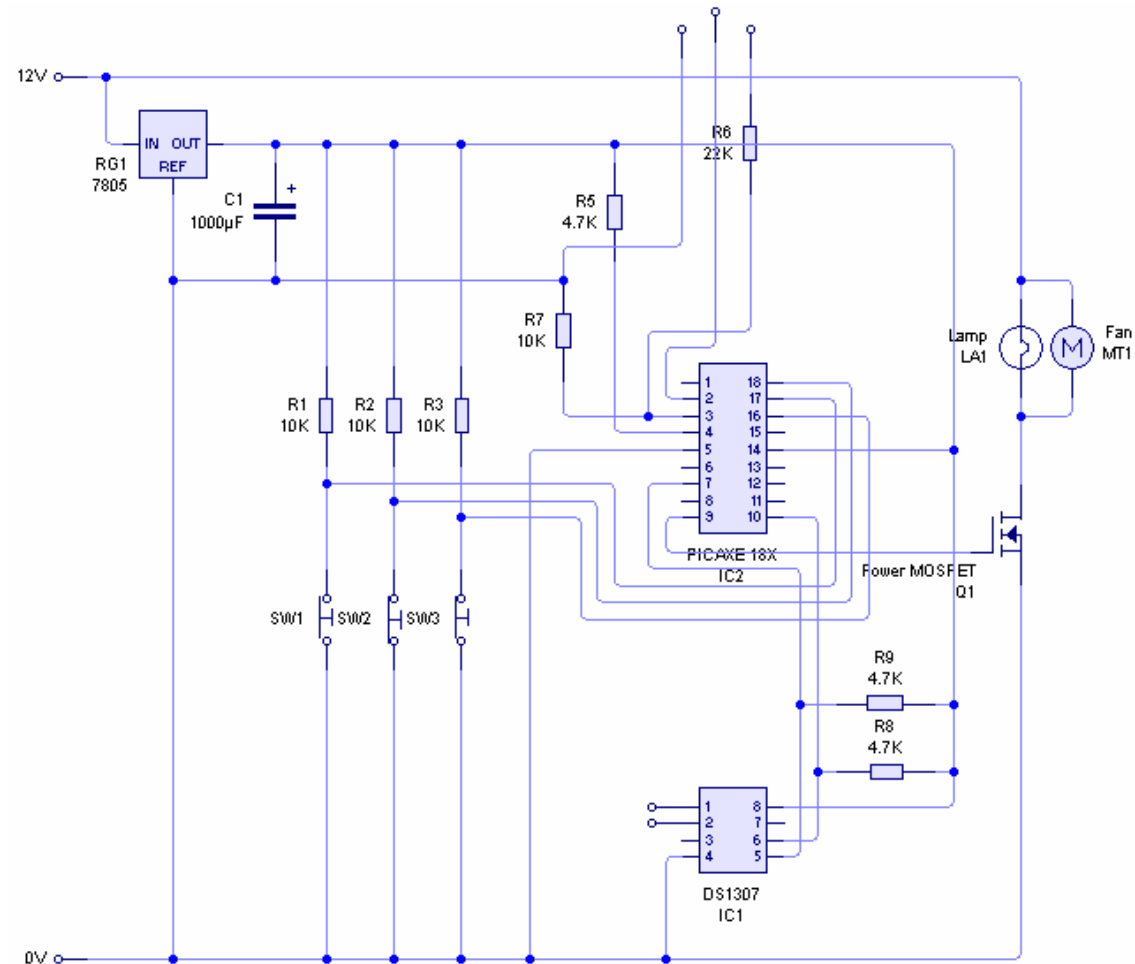
The schematic of this new setup is shown on the right - I created this using a circuit simulation package called Livewire.

Not shown on the schematic is the LCD and its connection to the PICAXE (the serial in wire from the LCD was connected to pin 6 of the PICAXE).

The gate of Power MOSFET, Q1 (order code 47-0400 from Rapid Electronics) is connected to pin 9 of the PICAXE because the 18X components only support pwmout on this pin.

Pins 1 and 2 of the DS1307 do not appear connected to anything in the schematic; they are actually connected to the 32.768kHz oscillator, it's just that Livewire does not have one available.

Also note that the picture of the breadboard setup and this schematic show a 4k7 pull up resistor both the I2C clock and data lines; I forgot to include these initially which led to a minor problem.



Schematic created with Livewire of PICAXE 18X and DS1307 (LCD not shown - it was connected on pin 6 for serial communication)

## Development - Breadboarding & Programming

---

```

1  init:
2      ' Let the LCD firmware initialise
3      pause 500
4
5      ' Set up the i2c bus
6      i2cslave %11010000, i2cslow, i2cbyte
7
8  main:
9      ' Reads data from clock chip
10     readi2c 0, (b0,b1,b2,b3,b4,b5,b6)
11
12     ' Write time and date to screen
13     serout 0,N2400,(254,1,"The time
14     is:",254,192,#b4,"/",#b5,"/",#b6,"
15     ",#b2,":",#b1,":",#b0)
16
17     ' Pause
18     pause 1000
19
20     ' Loop back to start
21     goto main

```

Shown on the left is the program I wrote to test this new setup, so I could be sure it worked before proceeding with the development of the rest of the program.

Unlike the previous program this one is considerably shorter because I haven't included the as yet unused symbol definitions for the push to break switches. I also did not bother with symbol definitions for the byte variables - because the program is so short they are not really necessary.

There is also no longer a "clear" subroutine because I realised that the LCD can be cleared in the same command as writing to it, simply by adding the '254,1' command sequence to the start of the write command sequence. This helps significantly with reducing the size of both the source and compiled program.

The final thing that allows this program to be much shorter is that because only one I2C device is used (the DS1307), I now only need one i2c slave command which is executed in the initialisation block.

### Note

The fourth byte register of the DS1307 is the day value and is taken with Sunday being the first day of the week - so a value of \$02 (the \$ means it is a binary coded decimal) means the day is Monday.

I may use this value later when I write the rest of the program, so that the program knows whether it the weekend or not (if it is equal to 1 or 7).

This will enable to me to add an additional feature to the alarm clock; the option of whether or not to run the wake-up call at weekends.

## Development - Breadboarding & Programming

---

Before my test program was of any use I had to actually program a time into the DS1307 and so modified the original program I wrote which used serial out. The amended program is shown below:

```
1  main:
2      i2cslave %11010000, i2cslow, i2cbyte
3      writei2c 0, ($00, $01, $21, $02, $05, $02, $07, $10)
4  end
```

The numbers in the writei2c command represent the following:  
seconds, minutes, hours, day, date, month, year, control

This is necessary because I had omitted connecting the 3v backup cell to the DS1307 when moving it to breadboard, so every time power is removed it resets. Having run this program to set the time, I then downloaded my simplified test program to the PICAXE and let it run. Surprisingly enough, it didn't work. Bit of a trend developing here... Everything was being output as 255 on the LCD:

```
The time is:
255/255/255 255:255:255
```

This was not the only visible problem. Watching the screen, every second (every update), the characters would noticeably blink in sequence scrolling from left to right, starting on the top row then moving across the bottom row. I immediately realised this was due to a latency issue somewhere; either the command to clear the screen, or command sequence to write the new data to it was taking so long that the update was noticeable. I remembered reading in the AXE033 and 034 datasheet that time must be allowed for the clear command to process - specifically 30ms - therefore this "blinking" problem can be attributed to the clear command (including the 30ms wait is not an option because this would mean the screen would be blank for a period of time).

After having thoroughly checked everything I realised that in moving the DS1307 from being on the LCD module to being separate on the breadboard, I had forgotten to connect a 4k7 pull-up resistor on each of the I2C clock and data lines. Connecting these fixed the problem and gave the output shown top right (which is meant to be in the format date/month/year hours:minutes:seconds). The PICAXE hadn't been communicating with DS1307 at all until I had added the pull-up resistors, so I ran the time setting program again and then ran the test program.

## Development - Breadboarding & Programming

### Result!

The picture, top right, shows the output of this new circuit and program.

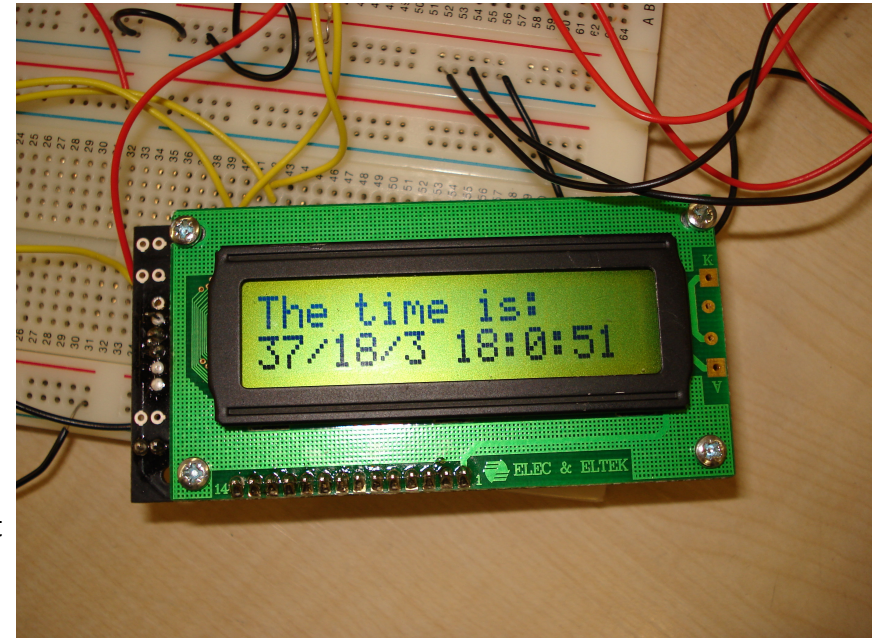
I managed to solve the scrolling effect problem by removing the clear command entirely - it was actually unnecessary (provided that the next string written is long enough to overwrite any values remaining on screen from the last).

Admittedly it may be wrong in thinking it's the 37th day of the 18th month of the 3rd year, and the minutes ought to really read '00' not '0' but this is definitely progress!

However, the problem is deeper than this; I didn't set the time to 18:00 as it reports - it was more like 12:30 midday, during a lesson. The second is perhaps the only value that appear correct. Yet, when watching the values change it is the seconds that are most strikingly wrong.

The reason for this is that the seconds, naturally, update most frequently. This makes the depth of the problem more obvious because the pattern (or more specifically lack of) that the value of the second field follows can be watched without having to wait minutes/hours/days/months/years each time.

It took me several attempts to spot any pattern in the values.



Output of the new setup.

## Development - Breadboarding & Programming

---

### Spot the pattern!

The seconds, although changing value every second, were not the correct values. These are my observations:

- They appeared to be skipping values (e.g. it would go from reading 41 seconds to 48 seconds the next), and counting up to a maximum of 89 seconds!
- After watching for a few minutes I noticed that it appeared to skip the same values every time.
- I then timed how long it took for the seconds value to return to 0 (it started on zero), and found it to be one minute exactly.

Though observation this confirmed my suspicion - that the values were not random - it did not help advance my attempts to work out the cause of the problem. In a further attempt to find the pattern, I wrote down every value it showed in a minute and the order they appeared in. It was as follows:

<i>Second</i>	<i>Value</i>	<i>Second</i>	<i>Value</i>	<i>Second</i>	<i>Value</i>	<i>Second</i>	<i>Value</i>	<i>Second</i>	<i>Value</i>	<i>Second</i>	<i>Value</i>
0	0	10	16	20	32	30	48	40	64	50	80
1	1	11	17	21	33	31	49	41	65	51	81
2	2	12	18	22	34	32	50	42	66	52	82
3	3	13	19	23	35	33	51	43	67	53	83
4	4	14	20	24	36	34	52	44	68	54	84
5	5	15	21	25	37	35	53	45	69	55	85
6	6	16	22	26	38	36	54	46	70	56	86
7	7	17	23	27	39	37	55	47	71	57	87
8	8	18	24	28	40	38	56	48	72	58	88
9	9	19	25	29	41	39	57	49	73	59	89

# Development - Progress Review

---

**06/02/07**

I've been aware that for the last couple of weeks I have been behind schedule according to my time plan; by now I should be on PCB production and yet I'm struggling with the development of my idea, having had quite a few unanticipated problems with it. I decided to include this progress review for two reasons:

- 1) To form a summary of what I have achieved so far.
- 2) To recognise the problems I must overcome before I can proceed with the project.

## Achievements

- Designed the ideal system diagram.
- Made working demo board and program for LCD Module.
- Correctly interfaced power MOSFET and Fan with PICAXE.
- Discovered that numbers from variables cannot be printed to LCD when using I2C.
- Subsequently recognised the need to separate the clock and LCD, so I2C can be used for clock and serial out for LCD.
- Correctly interfaced DS1307 clock IC with PICAXE via I2C.
- Discovered components of the time and date are printed incorrectly.
- Decided to use case idea one for the product casing.

## Problems

1. Need to fix the time and date printing.
2. Need to do research to find an appropriate sound output for when the assertive part of the wake-up call begins.
3. Need to find some momentary action digital inputs for the menu system (currently using PTB switches on breadboard for program development).
4. Need to develop the lamp override dimmer part of the system.
5. Need to find an appropriate power supply for the finished product (having to plug it into a bench power supply is not ideal; additionally, the bench power supplies are not capable of providing the current for a 100w lamp).
6. Need to get permission to use a 50w bulb and Delta 80mm fan instead of the insufficient 20W bulb and generic fan (which outputs 40CFM as opposed to the 80CFM of the delta - more importantly though I don't believe the breeze created would wake anyone up).

## Development - A Programming Revelation

---

Exhausted of ideas for what could be the problem with my program, jumping from 41 seconds to 48 seconds etc. I decided to try an alternative method of problem solving; see if anyone else had had similar problems. I checked the PICAXE user forums and searched for threads with the keyword 'DS1307'. After trawling through pages of irrelevant results I found a gem of a thread containing the following sample program for using the DS1307:

```

; Example of how to use DS1307 Time Clock (i2c device)
; Note the data is sent/received in BCD format.

symbol seconds = b0
symbol mins = b1
symbol hour = b2
symbol day = b3
symbol date = b4
symbol month = b5
symbol year = b6
symbol control = b7
symbol temp = b8

' set DS1307 slave address
i2cslave %11010000, i2cslow, i2cbyte

' uncomment this line to update the clock time
' goto start_clock

' read time and date and display on serial LCD module
init:
serout 7,N2400,(254,1) 'clear LCD
pause 30
main:
readi2c 0,
(seconds,mins,hour,date,month,year)
'debug b0 '(optional debug computer to screen)

serout 7,N2400,(254,192)
let temp = date & %00110000 / 16
serout 7,N2400,(#temp)
let temp = date & %00001111
serout 7,N2400,(#temp,"/")

let temp = month & %00001000 / 16
serout 7,N2400,(#temp)
let temp = month & %00001111
serout 7,N2400,(#temp,"/")

let temp = year & %11110000 / 16
serout 7,N2400,(#temp)
let temp = year & %00001111
serout 7,N2400,(#temp," ")

serout 7,N2400,(254,128)

let temp = hour & %00110000 / 16
serout 7,N2400,(#temp)
let temp = hour & %00001111
serout 7,N2400,(#temp,":")

let temp = mins & %01110000 / 16
serout 7,N2400,(#temp)
let temp = mins & %00001111
serout 7,N2400,(#temp,":")

let temp = seconds & %01110000 / 16
serout 7,N2400,(#temp)
let temp = seconds & %00001111
serout 7,N2400,(#temp)

pause 100
goto main

```



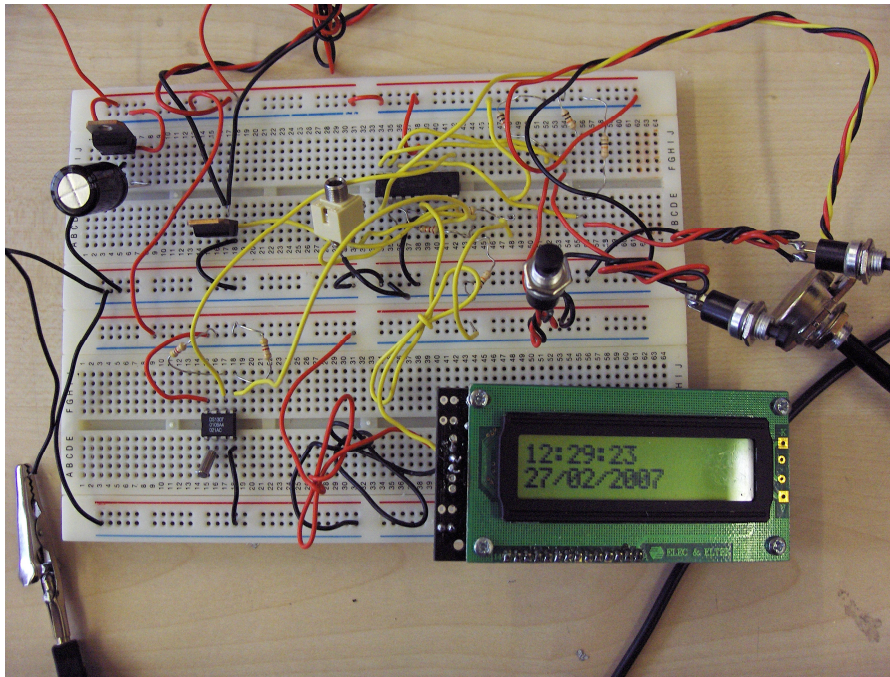
## Development - A Programming Revelation

```
'write time and date e.g. to 11:59:00 on Thurs 25/12/03
start_clock:
let seconds = $00 ' 00 Note all BCD format
let mins = $59 ' 59 Note all BCD format
let hour = $11 ' 11 Note all BCD format
let day = $03 ' 03 Note all BCD format
let date = $25 ' 25 Note all BCD format

let month = $12 ' 12 Note all BCD format
let year = $03 ' 03 Note all BCD format
let control = %00010000 ' Enable output at 1Hz

writei2c 0,
(seconds,mins,hour,day,date,month,year,control)
goto main

end
```



Finally, correct time and date output!

The fact that the program was written to output via serout confirms what I had thought regarding writing to the LCD with I2C; it is presumably not possible as I would expect the above sample program to be written for I2C if it were otherwise.

I had to modify the program so that the serout commands output to output 0 instead of output 7 as shown in the above program.

Having done this, I downloaded and ran the program on the PICAXE. Success, it worked! The screen output is shown on the left.

Note that the date is considerable later than you might expect at this stage in development - this is because I took the photo quite a while after this stage in development, and did not think to correct the date for the sake of documentation. Also note the addition of a potentiometer to the breadboard; by the time I had taken this photo I had started work on the lamp override dimmer.

## Development - False Assumptions

---

In order to proceed with the development of the rest of the program, it was crucial that I understand what had caused the 'number jumping' problem, and how this sample program that I found fixed it. The best way of explaining this is to start by examining the assumptions I had made, and what was wrong with them.

### Not Pure Binary Encoding

I had assumed all along that the values stored in the byte registers of the DS1307 were encoded as pure binary. The simplest way of clarifying what I mean here is with a few examples.

Let's say that the second should read as 45. If the values actually were stored in the DS1307 in pure binary the seconds register would read:

**00101101**

This is shown by the table below:

									<i>Totals</i>
<b>Bit Value</b>	128	64	32	16	8	4	2	1	255
<b>Bit Status</b>	0	0	1	0	1	1	0	1	n/a
<b>Decimal Value</b>	0	0	32	0	8	4	0	1	45

However when the programs I had written read 00101101 from the seconds byte register of the DS1307, 45 was not the number printed to the screen. Of course, I didn't know what binary value corresponded with the 45th second but that is not the point.

# Development - False Assumptions

---

## Binary Coded Decimals

I was wrong in assuming the values contained in the DS1307 byte registers are pure binary; they are actually stored as binary coded decimals.

Up until now I had not realised the significance of prefixing the values with a \$ when writing data to the DS1307, I'd just done it anyway because that is what is done in the examples in the AXE033 and 034 datasheet. For example:

```
writei2c 0, ($00, $01, $21, $02, $05, $02, $07, $10)
```

I was aware that the \$ character tells the compiler that the value following it is a binary coded decimal but simply hadn't thought about what this meant.

Extract from the Wikipedia article on Binary Coded Decimals (see appendix A for full article and citation):

In computing and electronic systems, Binary-coded decimal (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of circuits needed to implement mathematical operations and a relatively inefficient encoding – 6 wasted patterns per digit. Even though the importance of BCD has diminished [citation needed], it is still widely used in financial, commercial, and industrial applications.

...(continued)...

To BCD-encode a decimal number using the common encoding, each digit is encoded using the four-bit binary bit pattern for each digit. For example, the number 127 would be:

```
0001 0010 0111
```

Since most computers store data in eight-bit bytes, there are two common ways of storing four-bit BCD digits in those bytes:

## Development - False Assumptions

---

each digit is stored in one byte, and the other four bits are then set to all zeros, all ones (as in the EBCDIC code), or to 0011 (as in the ASCII code)

two digits are stored in each byte.

Unlike pure binary encodings large numbers can easily be displayed by splitting up the nibbles and sending each to a different character with the logic for each display being a simple mapping function. Converting from pure binary to decimal for display is much harder involving integer multiplication or divide operations. The BIOS in many PCs keeps the date and time in BCD format, probably for historical reasons (it avoided the need for binary to ASCII conversion).

...(continued)...

BCD is very common in electronic systems where a numeric value is to be displayed, especially in systems consisting solely of digital logic, and not containing a microprocessor. By utilising BCD, the manipulation of numerical data for display can be greatly simplified by treating each digit as a separate single sub-circuit. This matches much more closely the physical reality of display hardware—a designer might choose to use a series of separate identical 7-segment displays to build a metering circuit, for example. If the numeric quantity were stored and manipulated as pure binary, interfacing to such a display would require complex circuitry. Therefore, in cases where the calculations are relatively simple working throughout with BCD can lead to a simpler overall system than converting to 'pure' binary.

The same argument applies when hardware of this type uses an embedded microcontroller or other small processor. Often, smaller code results when representing numbers internally in BCD format, since a conversion from or to binary representation can be expensive on such limited processors. For these applications, some small processors feature BCD arithmetic modes, which assist when writing routines that manipulate BCD quantities.

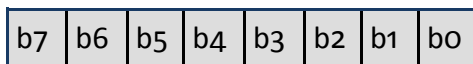
## Development - False Assumptions

---

This is the key to the whole problem. Using the 45th second as an example, I shall run through everything to clarify things.

When it actually is the 45th second, the seconds register of the DS1307 contains the value 45 as a binary coded decimal in it. For the purposes of this example it is necessary to convert 45 to BCD format. To start with, the number is split into its component digits; 4, and 5.

As the Wikipedia article states, two digit binary coded decimals may be stored in the nibbles of a single byte. This is true with the DS1307. Below is a visualisation of a byte (I have labelled the bits b0 through b7 - this labelling is not to be confused with the names of the byte variables used in BASIC programming):



The way data is stored as binary coded decimals in the DS1307 is illustrated below:



As can be seen the byte is split into nibbles and in these nibbles, the values 4 and 5 are stored. As one would expect, the 4 (tens) is stored in the most significant bits (b4 to b7) and the 5 (units) is stored in the least significant bits (b0 to b3).

Decimal 4 in binary is **0100** and decimal 5 in binary is **0101**. It should be clear therefore that decimal 45, when stored as a binary coded decimal in a single byte, will be **01000101** as the two nibbles are just joined together to form a byte.

To work out the value stored in it, process I have just described is reversed; the byte is split into two nibbles, and the decimal value of the left hand side nibble is the tens while that of the right hand side nibble is the units.

It is interesting to find the decimal value when this byte is assumed to be pure binary encoding, since this is what I had programmed the PI-CAXE to do and hence why the seconds were exhibiting that 'jumping' phenomena I described.

## Development - False Assumptions

---

### Pattern found!

I created a set of tables which show for the range 0 to 59, the binary equivalent of the binary coded decimal seconds value. From this, I then calculated the decimal interpretation when the binary is assumed to be pure (as opposed to binary coded decimals).

Rather than calculating these values by hand with the table conversion system I used for illustrational purposes earlier, I used the Calculator program included with Microsoft Windows; in scientific mode it supports hex, dec, oct and bin notation; to convert BCD to bin it was simply a case of entering the BCD value in hex mode (hex and BCD are one and the same in this context), then changing the view mode to bin. For the decimal value I just changed the view mode to dec. I have inserted spaces between the nibbles so that it is easier to recognise the pattern as the units and tens increase.

<i>BCD</i>	<i>Bin</i>	<i>Dec</i>	<i>BCD</i>	<i>Bin</i>	<i>Dec</i>	<i>BCD</i>	<i>Bin</i>	<i>Dec</i>
0	0000 0000	0	10	0001 0000	16	20	0010 0000	32
1	0000 0001	1	11	0001 0001	17	21	0010 0001	33
2	0000 0010	2	12	0001 0010	18	22	0010 0010	34
3	0000 0011	3	13	0001 0011	19	23	0010 0011	35
4	0000 0100	4	14	0001 0100	20	24	0010 0100	36
5	0000 0101	5	15	0001 0101	21	25	0010 0101	37
6	0000 0110	6	16	0001 0110	22	26	0010 0110	38
7	0000 0111	7	17	0001 0111	23	27	0010 0111	39
8	0000 1000	8	18	0001 1000	24	28	0010 1000	40
9	0000 1001	9	19	0001 1001	25	29	0010 1001	41

(tables continued overleaf)

## Development - False Assumptions

---

### Pattern matched!

<i>BCD</i>	<i>Bin</i>	<i>Dec</i>	<i>BCD</i>	<i>Bin</i>	<i>Dec</i>	<i>BCD</i>	<i>Bin</i>	<i>Dec</i>
30	0011 0000	48	40	0100 0000	64	50	0101 0000	80
31	0011 0001	49	41	0100 0001	65	51	0101 0001	81
32	0011 0010	50	42	0100 0010	66	52	0101 0010	82
33	0011 0011	51	43	0100 0011	67	53	0101 0011	83
34	0011 0100	52	44	0100 0100	68	54	0101 0100	84
35	0011 0101	53	45	0100 0101	69	55	0101 0101	85
36	0011 0110	54	46	0100 0110	70	56	0101 0110	86
37	0011 0111	55	47	0100 0111	71	57	0101 0111	87
38	0011 1000	56	48	0100 1000	72	58	0101 1000	88
39	0011 1001	57	49	0100 1001	73	59	0101 1001	89

From the tables it can be seen that no 'jumping' occurs for BCD values with the same tens value; it is only when the tens value increases that jumps occur in the decimal equivalent, and every time it is an increase of 7 (it is interesting to note that 7 is the difference between 16 and 9, where 16 is the range of values expressible in a single nibble and 9 is the highest number expressible in a single character as a decimal, though I shall not explain the significance as it is beyond the scope of this documentation and is not important to the creation of my alarm clock!).

When the full set of decimal interpretations on BCD values from the above tables is compared to the set of numbers I observed the LCD cycle through when I was interpreting the DS1307 registers as pure binary, they match up exactly. In other words, the seconds column of my observations table match up with the BCD column of the above table, and the value column of the observation table matches up with the dec column of the above tables I created.



# Development - BCDs and Programming

---

## Advantage of BCDs

As stated in the Wikipedia article I quoted on binary coded decimals, the advantage of BCDs is that they simplify the printing of numbers to displays. This is because they remove the requirement for padding. What I mean by this is best illustrated by a picture - the display reads:



As can be seen the minutes are printed as "0". This simply is not normal for timekeeping devices; it should be printed "00". The same applies if the value were 4 - it should be printed "04" not "4". If the data actually was stored in the DS1307 registers as pure binary, I would have had to develop a way of adding padding where necessary.

I think this would actually have been quite simple - just a case of checking the value to be printed before it is, and if it's less than 10, going to a subroutine that prints a 0 to the display before returning and printing the single unit character. It is the fact that the tens and units are stored as separate values that removes the need for this with BCDs.

## Disadvantage of BCDs

The Wikipedia article on BCDs also states;

Some operations are more complex to implement. Adders require extra logic to cause them to wrap and generate a carry early. 15%-20% more circuitry is needed for BCD add compared to pure binary.

as I was to discover later when developing the setup menu system. The same also applies for subtraction, as I also discovered when writing the setup menu system.

## Development - BCDs and Programming

---

Now that I knew what the problem was, I needed to know how the sample program I found worked around it. The answer lies in a block of code that occurs several times in the program:

```
let temp = seconds & %01110000 / 16
serout 7,N2400,(#temp)
let temp = seconds & %00001111
serout 7,N2400,(#temp)
```

It is reasonable obvious that the above block prints the seconds to screen based on the fact that it reads information from the variable with symbol 'seconds'.

Since I now know how the data is stored in the DS1307, it is reasonable simple to interpret the above block since I already know what it does.

The data contained in the seconds variable is the seconds value in BCD format. In order to print it to the LCD, the byte must be split into two to obtain the tens and units values, which can then be printed to the LCD.

The first line

```
let temp = seconds & %01110000 / 16
```

contains several things I hadn't encountered before in BASIC. Together these work to extract the tens value from the seconds variable into the temp variable which is then printed to the LCD on the second line. The first part is quite simple; the "let temp = " part is an assignment operator which loads the return value of the expression to the right into the byte variable temp - simple enough.

## Development - BCDs and Programming

---

### Stepping through the code

Examining the right hand side more closely, I have labelled the different parts of the code:

<code>seconds</code>	<code>&amp;</code>	<code>%01110000</code>	<code>/</code>	<code>16</code>
byte variable	bitwise AND operator	binary constant used as bitmask (% instructs compiler that it is in binary notation)	divide operator	divisor

This example should make it clear what the code does. I shall use an actual seconds value of 45 again for this example. From the conversions tables I created, a seconds value of 45 is

```
0100 0101
```

in binary. Before I continue, bitmasks and bitwise operators need to be introduced. From the Wikipedia article on masks:

In computer science, a mask is some data that, along with an operation, is used in order to extract information stored elsewhere.

The most common mask used, also known as a bitmask, extracts the status of certain bits in a binary string or number. For example, if we have the binary string 10011101 and we want to extract the status of the fifth bit counting along from the most significant bit, we would use a bitmask such as 00001000 and use the bitwise AND operator. Recalling that 1 AND 1 = 1, with 0 otherwise, we find the status of the fifth bit, since

```
10011101 AND 00001000 = 00001000
```

Likewise we can set the fifth bit by applying the mask to the data using the OR operator.

...(continued)...

## Development - BCDs and Programming

You can also use bitmasks to easily check the state of individual bits regardless of the other bits. To do this, you simply turn off all the other bits using the bitwise AND as discussed above and see if the resulting value is 0. If it is, then the bit was off, but if the value is any other value, then the bit was on. What makes this so convenient is that you do not need to figure out what the value actually is, you just need to know that it is not 0.

Substituting the binary seconds value back into the line gives:

```
let temp = %01000101 & %01110000 / 16
```

(Note I have used a % to denote that it is in binary notation).

I have created a table to illustrate the function of the bitwise AND operator:

Left Hand Value	0	1	0	0	0	1	0	1
Right Hand Value	0	1	1	1	0	0	0	0
Return Value	0	1	0	0	0	0	0	0

As can be seen this gives a return value of

```
01000000
```

which as a decimal is 64. The final operator in the line is the division operator, which divides the return value of the bitwise AND operator by 16:

$$64 / 16 = 4$$

Recalling that the seconds value I used in this example was 45, this shows that the first line will load the tens value (4) into the temp variable and print it to the LCD.

## Development - BCDs and Programming

---

Running through the same procedure but this time for the third line. Notice the bitmask is different - it masks off the tens instead, returning the units - and there is no division by 16:

```
let temp = %01000101 & %00001111
```

Left Hand Value	0	1	0	0	0	1	0	1
Right Hand Value	0	0	0	0	1	1	1	1
Return Value	0	0	0	0	0	1	0	1

The binary return value

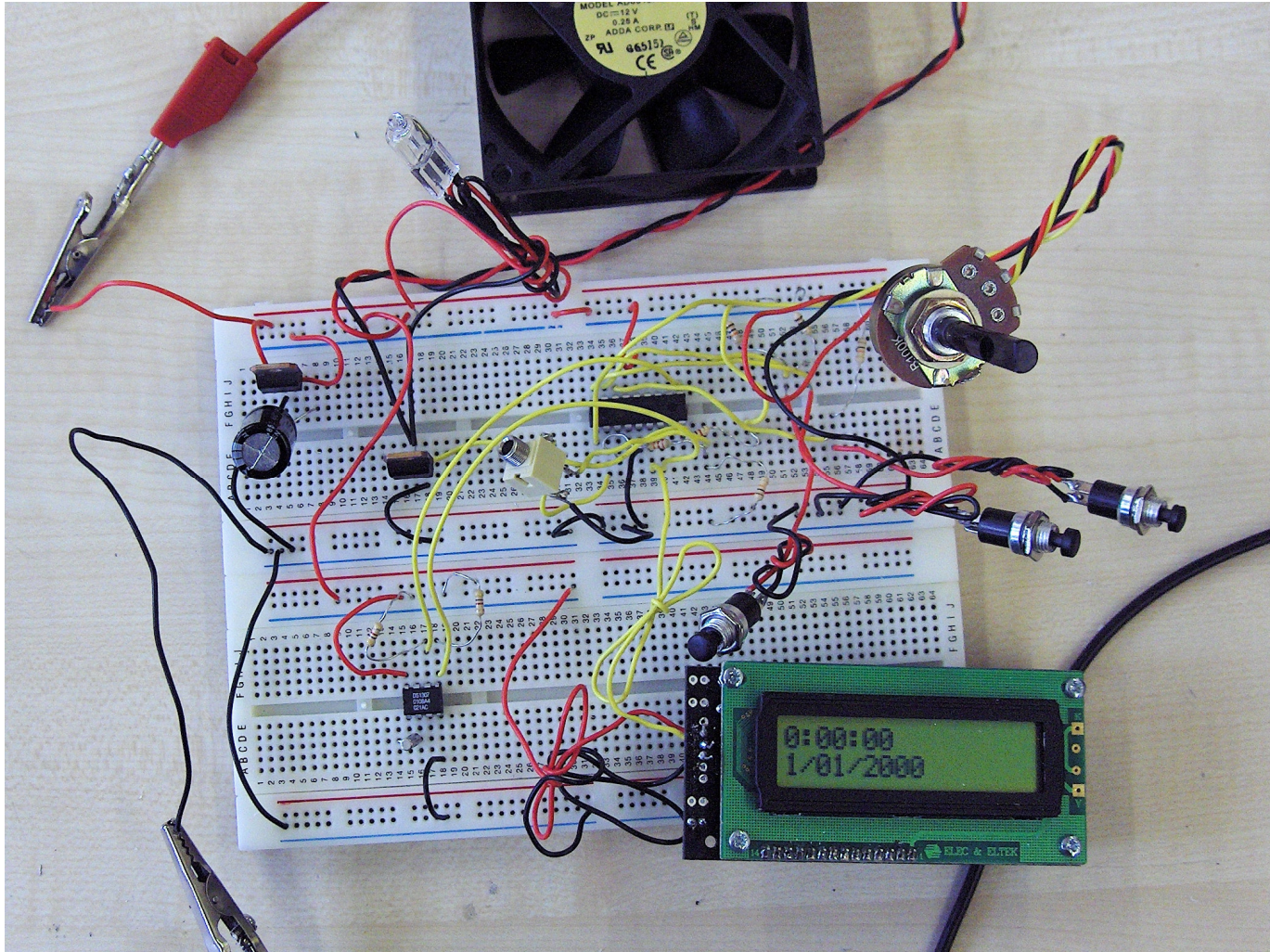
```
00000101
```

is 5 as a decimal.

Recalling that the seconds value was 45, that the first two lines of the code block printed 4 to the screen, and the second two lines print 5 to the screen, I have demonstrated how the sample program works, and in doing so ensured that I fully understand the process.



## Development - Lamp Dimming



Breadboard with 100k potentiometer for lamp dimming

For the sake of completeness I have included, shown on the left, a picture of the breadboard with the 100k potentiometer I connected to the PICAXE analogue 1 in pin (physical pin 18). I connected the ends of the potentiometer to +5v and 0v.

The programming of the dimming function is detailed in 'Putting it all together'.

## Development - Putting it all together (Programming)

After several weeks of slow progress I had finally managed to get to the point of writing the fully featured program. Knowing that the program would be very long and quite complicated compared to those I had already written, I decided the best way of setting about this task would be to break it into smaller components.

### Program components

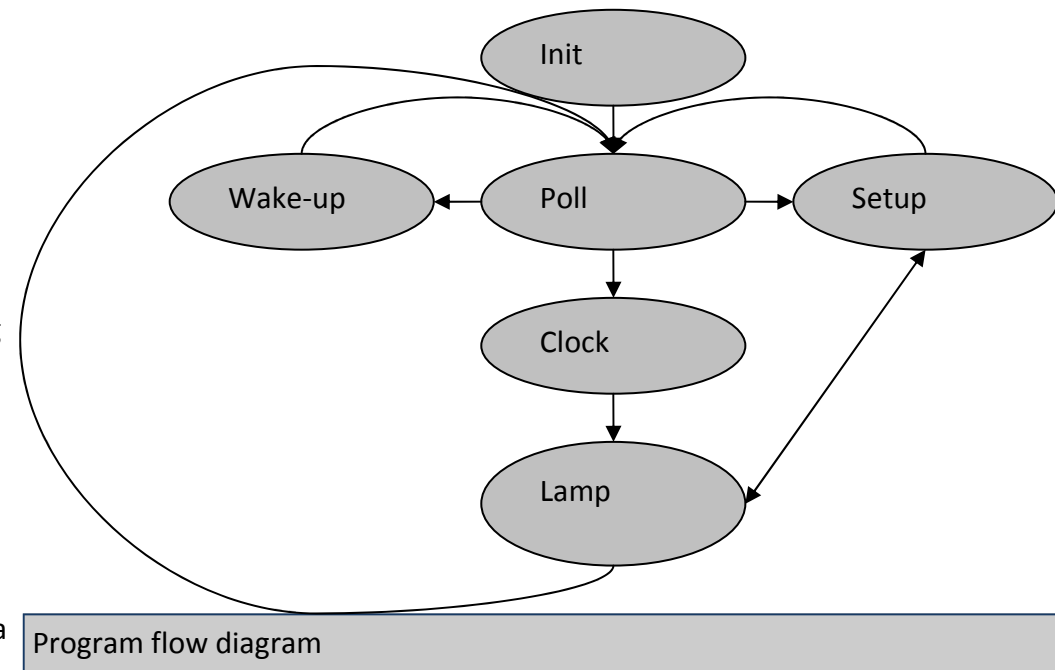
1. Initialisation
2. Polling
3. Time display
4. Setup menus
5. Lamp override
6. Wake-up call

### Program flow

Shown on the right is a diagram of the program flow, showing how the different program components relate to each other. Below follow descriptions of the function of each program component.

### Initialisation

A lot of it is obvious; there is the familiar 500ms pause to allow the LCD driver IC to initialise. I have however added, just after the 500ms wait, a clear command so that the display is a blank canvas prior to anything being printed to it followed by the required 30ms wait.



I avoided using the clear command anywhere else in the program because of the problem I encountered previously in development. This was because the clear command requires a 30ms wait to allow clearing to complete, and had caused the unwanted scrolling effect prior to me removing it. However, a 30ms wait at the start of the program is not a problem and it is useful to have it there so I can be certain the display is



## Development - Putting it all together (Programming)

---

blank when the program starts.

The setup of the I2C bus for communication with the DS1307 is also familiar because I have done so before, and the same applies for the input symbols (though this program will be the first time I use them).

The part of the init stage that required the most thought was planning which variables to store what data in. The PICAXE has a total of 14 byte variables available. In order to assign this efficiently I had to think ahead work out what information I would need to have available in the program. The init block is shown below:

```

1  init:
2      ' Let the LCD Firmware initialise and clear the
screen
3      pause 500
4      serout 0,N2400,(254,1)
5      pause 30
6
7      ' Set up the i2c bus
8      i2cslave %11010000, i2cslow, i2cbyte
9
10     ' Memory locations
11     symbol seconds = b0
12     symbol mins = b1
13     symbol hour = b2
14     symbol day = b3
15     symbol date = b4
16     symbol month = b5
17     symbol year = b6
18     symbol control = b7
19     symbol alarmmin = b8
20     symbol alarmhour = b9
21     symbol fullweekalarm = b10
22     symbol fademins = b11
23     symbol dismissmins = b12
24     symbol temp = b13
25
26     ' Set up the input symbols
27     symbol UP = pin0
28     symbol DOWN = pin1
29     symbol SET = pin7

```

## Development - Putting it all together (Programming)

When planning my usage of the available byte variables I had to consider what information I would need available in the program. While on the topic of memory management, it is appropriate to mention the register locations of the DS1307. Shown on the right is a diagram, copied from the DS1307 datasheet, of the DS1307 registers.

It can now be seen quite clearly that the values are stored as binary coded decimals, shown by the splitting of the bytes into nibbles.

### Polling

The polling section of the program is the main section that every other program component returns to when complete. The polling section performs three main tasks:

1. Polls for SET button being pressed and if true jumps to the setup menu component.
2. Checks if the wake-up call should be run (i.e. is it time to wake up).
3. Jumps to the clock section if neither of the above are true, which prints the time and date to the screen.
4. Update the lamp brightness based on the position of the lamp dimmer potentiometer.

BIT7								BIT0		
00H	CH	10 SECONDS				SECONDS				00-59
	0	10 MINUTES				MINUTES				00-59
	0	12/24	10 HR A/P	10 HR		HOURS				01-12 00-23
	0	0	0	0	0	DAY				1-7
	0	0	10 DATE			DATE				01-28/29 01-30 01-31
	0	0	0	10 MONTH		MONTH				01-12
	10 YEAR				YEAR				00-99	
07H	OUT	0	0	SQWE	0	0	RS1	RS0		

Register location table

```

1 poll:                                     11 ' Poll for setup button press
2 ' Poll for lamp level change             12 if SET = 1 then setup
3 gosub lamp                               13
4                                           14 ' Is it the weekend?
5 ' Reads data from clock chip             15 'IF b3=7 OR b3=1 THEN
6 readi2c 0,                               16 ' IF b10 = 1 THEN
(seconds,mins,hour,day,date,month,year,control,alarmmin, 17 '
alarmhour,fullweekalarm,fademins,dismissmins) wakeup
7                                           18 ' ENDIF
8 ' Ensure control value is correct (not user 19 'ELSE
configurable)                              20 ' IF b1 = b8 AND b2 = b9 THEN goto wakeup
9 let control = 16                          21 'ENDIF
10                                           22

```

## Development - Putting it all together (Programming)

```

23     if day = 1 then ifallweek
24     if day = 7 then ifallweek
25
26 iftime:
27     if hour != alarmhour then clock
28     if mins != alarmmin then clock
29     goto wakeup
30
31 ifallweek:
32     if fullweekalarm = 1 then iftime

```

I have described the logic of the polling block that checks whether it is time to wake up in more detail below:

Because I have implemented the option of disabling the wake-up call at weekends, the first stage in determining whether the wake-up call is checking whether it is the weekend, and if it is, checking if the alarm is enabled for weekends. If it is not the weekend, or it is the weekend and the alarm is enabled at weekends, the program then checks if the realtime hour is equal to the alarm hour. If this is true it then checks if the realtime minute is equal to the alarm minutes. If this is true it jumps to the wakeup section. I had originally written the wakeup logic as follows (notice it is commented out):

```

14     ' Is it the weekend?
15     'IF b3=7 OR b3=1 THEN
16     '     IF b10 = 1 THEN
17     '         IF b1 = b8 AND b2 = b9 THEN goto wakeup
18     '     ENDIF
19     'ELSE
20     '     IF b1 = b8 AND b2 = b9 THEN goto wakeup
21     'ENDIF

```

But was given a compile error when attempting to download the program to the PICAXE. After much experimentation I found that the Programming Editor wouldn't accept the syntax of the if statements and so I had to write it out at a more basic level as visible at the top of the page.

## Development - Putting it all together (Programming)

---

### Clock

This section of the program simply prints the time & date to the screen and is the default screen. The code is remarkably simple:

```

1  clock:
2      serout 0,N2400,(254,192)
3      gosub printdate
4      serout 0,N2400,(254,128)
5      gosub printtime
6      goto poll

```

This is because I have made use of subroutines for printing the date and time. This was necessary because without them my program exceeded the maximum program size of 2048 bytes, so I had to optimise it via the use of subroutines for repeated sections of the code (the printdate and printtime sections are used several times in the setup menus).

### Print Routines

Though not shown on the program flow diagram, I have included the print routines here to clarify what I described above:

```

1  printtime:
2      let temp = hour & %00110000 / 16
3      serout 0,N2400,(#temp)
4      let temp = hour & %00001111
5      serout 0,N2400,(#temp,":")
6
7      let temp = mins & %01110000 / 16
8      serout 0,N2400,(#temp)
9      let temp = mins & %00001111
10     serout 0,N2400,(#temp,":")
11
12     let temp = seconds & %01110000 / 16
13
14     serout 0,N2400,(#temp)
15     let temp = seconds & %00001111
16     serout 0,N2400,(#temp)
17     return
18  printdate:
19     let temp = date & %00110000 / 16
20     serout 0,N2400,(#temp)
21     let temp = date & %00001111
22     serout 0,N2400,(#temp,"/")
23
24     let temp = month & %00010000 / 16

```

## Development - Putting it all together (Programming)

```

25     serout 0,N2400,(#temp)
26     let temp = month & %00001111
27     serout 0,N2400,(#temp,"/")
28
29     let temp = year & %11110000 / 16
30     serout 0,N2400,("\'20\'",#temp)
31     let temp = year & %00001111
32     serout 0,N2400,(#temp," ")
33     return
34
35  printalarmtime:
36
37     let temp = alarmhour & %00110000 / 16
38     serout 0,N2400,(#temp)
39     let temp = alarmhour & %00001111
40     serout 0,N2400,(#temp,":")
41
42     let temp = alarmmin & %01110000 / 16
43     serout 0,N2400,(#temp)
44     let temp = alarmmin & %00001111
45     serout 0,N2400,(#temp," ")
46     return

```

These subroutines print to the current cursor position on the LCD. This is to ensure maximum flexibility of use throughout the rest of the program.

### Lamp Override

The lamp override subroutine is called by the poll section to update the brightness of the lamp based on the position of the lamp dimmer potentiometer. The code is as follows:

```

1  lamp:
2     readadc10 2,w0
3     pwmout 3,249,w0
4     return

```

The readadc10 command returns a 10 bit value (i.e. ranging from 0 to 1023 decimal) that represents the voltage at the pin specified. Because the value is 10 bit, it cannot be stored in a byte variable and instead must be stored in a word (two byte) variable. Because the lamp subroutine is called before the byte variables are updated from the DS1307 it doesn't matter that writing to w0 (word variable 0) will cause the values of b0 and b1 to be overwritten (the byte and word variables use the same memory space).

## Development - Putting it all together (Programming)

---

The `pwmout 3,249,w0` command is a little more complicated. The 3 specifies which pin to output on, while 249 is the period and `w0` means that the value of `w0` is used as the duty cycle.

The complexity was in matching the period to the range of values that the duty cycle variable can have (0 to 1023 as already described). In order to find out the necessary period value for the duty cycle to reach 100% when the dimmer is at full, I used the `pwmout` wizard dialogue available in the PICAXE Programming Editor software and experimented with the PWM frequency value until I found a value that was derived from a duty cycle of 1023.

I found that a desired PWM frequency value of 4000MHz resulted in a PWM command with 249 as the period and 1000 as the duty cycle (for 100% power) - as close to 1023 as I could get it.

### Wake-up Call

The wake-up call is as the name suggests the section of the program that switches the fan on and slowly increases the duty cycle of the PWM MOSFET over the period specified by the `fademins` byte value (as the name suggests, measured in minutes). It then keeps the outputs running at full intensity for the period specified by the `dismissmins` byte value (again measured in minutes). Because the program that I am documenting here is the one I wrote for the breadboard, there is no code for the sound output because there wasn't one. See the PCB program documentation for sound output code. The breadboard code is as follows:

```

1  wakeup:                                12  pause    w1
2  serout 0,N2400,(254,128,"Wake-up!")    13  pwmout 3,0,0
3                                          14
4  let b0 = 0                              15  serout 0,N2400,(254,1)
5  let w1 = fademins * 300                 16  goto poll
6  for b0 = 0 to 200
7  pwmout 3,49,b0
8  pause w1
9  next b0
10
11 let w1 = dismissmins * 1000

```

## Development - Putting it all together (Programming)

---

Again the code is fairly short. However there was little bit of calculation involved, for the intensity and timing.

“Wake-up!” is printed to the top line of the LCD, while the date, month and year remain on the bottom line (as they were written to the LCD by the clock routine).

The first loop is for the ‘gentle’ part of the wake-up call; it loops 200 times and each time a pwmout command is executed with the loop counter variable used as the duty cycle parameter. The number of loops is 200 for two reasons:

- 1) It must be less than 255 to avoid having to use a word variable - as I already described memory is a scarce resources when programming for the PICAXE.
- 2) 200 is a nice round number for use as the maximum duty cycle reached.

I then had to work out the necessary period of the pwmout command so that the intensity reaches maximum when the duty cycle is 200. I did this using the same technique I described on the previous page.

That was the first calculation for the ‘gentle’ wake-up call, concerning intensity. The second calculation was for the timing; the fademins variable contains the number of minutes during which to fade in. I used the pause command in the loop to implement a wait every time the loop runs. The period of a pause is measured in ms, so I needed to convert the fademins variable into a value that would equal 1/200 of 15 mins (1/200 because the loop runs 200 times). I formed an equation linking the pause parameter and the fademins parameter, where  $p$  is the pause value and  $f$  is the fademins value. I then rearranged it to get an equation for  $p$  in terms of  $f$ :

$$p \times 200 = f \times 60 \times 1000$$

$$200p = 60000f$$

$$p = \frac{60000f}{200}$$

$$p = 300f$$



## Development - Putting it all together (Programming)

---

The line:

```
let w1 = fademins * 300
```

preceding the gentle wake-up loop defines the pause value for use in the loop:

```
pause w1
```

It was necessary to use a word variable because even with a fademins of value the pause value is greater than 255! The maximum fademins value that can be set via the setup menu is 60, and  $60 \times 300 = 18000$  so the word variable is more than adequate for the range of values it must contain.

The second part of the wake-up call:

```
11     let w1 = dismissmins * 1000
12     pause    w1
13     pwmout 3,0,0
14
15     serout 0,N2400,(254,1)
16     goto poll
```

is the 'assertive' part of the wake-up call. The code is much simpler because all the program must do is wait for the number of minutes specified by the dismissmins variable - this is simply a case of multiplying the dismissmins value by 1000 and storing it in a word variable for use by the pause command. Once the waiting is complete the program switches off the outputs, clears the screen, and jumps to the polling block. Because the program I am documenting here is the breadboard version, there is no implementation of separate control for the fan and lamp, since I hadn't set up the necessary extra MOSFET.

## Development - Putting it all together (Programming)

---

### Setup

```

1  setup:
2      serout 0,N2400,(254,1)
3
4  sethour:
5      if hour > $23 then resethour
6
7      serout 0,N2400,(254,128,"Hour:",254,192)
8
9      gosub printtime
10
11     pause 100
12     if UP = 1 then inhour
13     if DOWN = 1 then dechour
14     if SET = 1 then setmins
15     goto sethour
16
17 inhour:
18     let temp = hour & %00001111
19     if temp = 9 then inhourtens
20     let hour = hour + $01
21     goto sethour
22 inhourtens:
23     let temp = hour & %00110000
24     let hour = temp + $10
25     goto sethour
26 dechour:
27     if hour = $00 then fullhour
28     let temp = hour & %00001111
29     if temp = 0 then dechourtens
30     let hour = hour - $01
31     goto sethour
32 dechourtens:
33     let temp = hour & %00110000
34     let hour = temp - $10
35     let hour = hour + $09
36     goto sethour
37 resethour:
38     let hour = $00
39     goto sethour
40 fullhour:
41     let hour = $23
42     goto sethour
43
44 setmins:
45     if mins > $59 then resetmins
46
47     serout 0,N2400,(254,128,"Minutes:",254,192)
48
49     gosub printtime
50
51     pause 100
52     if UP = 1 then incmins
53     if DOWN = 1 then decmins
54     if SET = 1 then setseconds
55     goto setmins
56

```

## Development - Putting it all together (Programming)

---

```

57 incmins:
58     let temp = mins & %00001111
59     if temp = 9 then incminstens
60     let mins = mins+ $01
61     goto setmins
62 incminstens:
63     let temp = mins & %01110000
64     let mins = temp + $10
65     goto setmins
66 decmins:
67     if mins = $00 then fullmins
68     let temp = mins & %00001111
69     if temp = 0 then decminstens
70     let mins = mins - $01
71     goto setmins
72 decminstens:
73     let temp = mins & %01110000
74     let mins = temp - $10
75     let mins = mins + $09
76     goto setmins
77 resetmins:
78     let mins = $00
79     goto setmins
80 fullmins:
81     let mins = $59
82     goto setmins
83
84 setseconds:
85     if seconds > $59 then resetseconds
86
87     serout 0,N2400,(254,128,"Seconds:",254,192)
88
89     gosub printtime
90
91     pause 100
92     if UP = 1 then incseconds
93     if DOWN = 1 then decseconds
94     if SET = 1 then setdayclear
95     goto setseconds
96
97 incseconds:
98     let temp = seconds & %00001111
99     if temp = 9 then incsecondstens
100    let seconds = seconds+ $01
101    goto setseconds
102 incsecondstens:
103    let temp = seconds & %01110000
104    let seconds = temp + $10
105    goto setseconds
106 decseconds:
107    if seconds = $00 then fullseconds
108    let temp = seconds & %00001111
109    if temp = 0 then decsecondstens
110    let seconds = seconds - $01
111    goto setseconds
112 decsecondstens:
113    let temp = seconds & %01110000
114    let seconds = temp - $10
115    let seconds = seconds + $09
116    goto setseconds
117 resetseconds:
118    let seconds = $00

```

## Development - Putting it all together (Programming)

---

```

119     goto setseconds
120 fullseconds:
121     let seconds = $59
122     goto setseconds
123
124 setdayclear:
125     serout 0,N2400,(254,1)
126
127 setdaydisp:
128     serout 0,N2400,(254,128,"Day:")
129     if day = $01 then sunday
130     if day = $02 then monday
131     if day = $03 then tuesday
132     if day = $04 then wednesday
133     if day = $05 then thursday
134     if day = $06 then friday
135     if day = $07 then saturday
136
137 sunday:
138     serout 0,N2400,(254,192,"Sunday ")
139     goto setdaypoll
140 monday:
141     serout 0,N2400,(254,192,"Monday ")
142     goto setdaypoll
143 tuesday:
144     serout 0,N2400,(254,192,"Tuesday ")
145     goto setdaypoll
146 wednesday:
147     serout 0,N2400,(254,192,"Wednesday")
148     goto setdaypoll
149 thursday:
150     serout 0,N2400,(254,192,"Thursday ")
151     goto setdaypoll
152 friday:
153     serout 0,N2400,(254,192,"Friday ")
154     goto setdaypoll
155 saturday:
156     serout 0,N2400,(254,192,"Saturday ")
157     goto setdaypoll
158
159 setdaypoll:
160     pause 100
161     if day > 7 then resetday
162     if day < 1 then fullday
163     if UP = 1 then incday
164     if DOWN = 1 then decday
165     if SET = 1 then setdate
166     goto setdaydisp
167
168 incday:
169     let day = day + 1
170     goto setdaydisp
171 decday:
172     let day = day - 1
173     goto setdaydisp
174 resetday:
175     let day = 1
176     goto setdaydisp
177 fullday:
178     let day = 7
179     goto setdaydisp
180

```

## Development - Putting it all together (Programming)

---

```

181 setdate:
182     if date > $31 then resetdate
183
184     serout 0,N2400,(254,128,"Date:   ",254,192)
185
186     gosub printdate
187
188     pause 100
189     if UP = 1 then incdate
190     if DOWN = 1 then decdate
191     if SET = 1 then setmonth
192     goto setdate
193
194 incdate:
195     let temp = date & %00001111
196     if temp = 9 then incdatetens
197     let date = date + $01
198     goto setdate
199 incdatetens:
200     let temp = date & %00110000
201     let date = temp + $10
202     goto setdate
203 decdate:
204     if date = $01 then fulldate
205     let temp = date & %00001111
206     if temp = 0 then decdatetens
207     let date = date - $01
208     goto setdate
209 decdatetens:
210     let temp = date & %00110000
211     let date = temp - $10
212
212     let date = date + $09
213     goto setdate
214 resetdate:
215     let date = $01
216     goto setdate
217 fulldate:
218     let date = $31
219     goto setdate
220
221 setmonth:
222     if month > $12 then resetmonth
223
224     serout 0,N2400,(254,128,"Month:  ",254,192)
225
226     gosub printdate
227
228     pause 100
229     if UP = 1 then incmonth
230     if DOWN = 1 then decmonth
231     if SET = 1 then setyear
232     goto setmonth
233
234 incmonth:
235     let temp = month & %00001111
236     if temp = 9 then incmonthtens
237     let month = month + $01
238     goto setmonth
239 incmonthtens:
240     let temp = month & %00010000
241     let month = temp + $10
242     goto setmonth

```

## Development - Putting it all together (Programming)

---

```

243 decmonth:
244     if month = $01 then fullmonth
245     let temp = month & %00001111
246     if temp = 0 then decmonthtens
247     let month = month - $01
248     goto setmonth
249 decmonthtens:
250     let temp = month & %00010000
251     let month = temp - $10
252     let month = month + $09
253     goto setmonth
254 resetmonth:
255     let month = $01
256     goto setmonth
257 fullmonth:
258     let month = $12
259     goto setmonth
260
261 setyear:
262     if year > $99 then resetyear
263
264     serout 0,N2400,(254,128,"Year:  ",254,192)
265
266     gosub printdate
267
268     pause 100
269     if UP = 1 then incyear
270     if DOWN = 1 then decyear
271     if SET = 1 then setalarmhour
272     goto setyear
273
274 incyear:
275     let temp = year & %00001111
276     if temp = 9 then incyeartens
277     let year = year + $01
278     goto setyear
279 incyeartens:
280     let temp = year & %11110000
281     let year = temp + $10
282     goto setyear
283 decyear:
284     if year = $00 then fullyear
285     let temp = year & %00001111
286     if temp = 0 then decyeartens
287     let year = year - $01
288     goto setyear
289 decyeartens:
290     let temp = year & %11110000
291     let year = temp - $10
292     let year = year + $09
293     goto setyear
294 resetyear:
295     let year = $00
296     goto setyear
297 fullyear:
298     let year = $99
299     goto setyear
300
301 setalarmhour:
302     if alarmhour > $23 then resetalarmhour
303
304     serout 0,N2400,(254,128,"Alarm Hour:

```

## Development - Putting it all together (Programming)

---

```

",254,192)
305
306     gosub printalarmtime
307
308     pause 100
309     if UP = 1 then incalarmhour
310     if DOWN = 1 then decalarmhour
311     if SET = 1 then setalarmmin
312     goto setalarmhour
313
314 incalarmhour:
315     let temp = alarmhour & %00001111
316     if temp = 9 then incalarmhourtens
317     let alarmhour = alarmhour + $01
318     goto setalarmhour
319 incalarmhourtens:
320     let temp = alarmhour & %00110000
321     let alarmhour = temp + $10
322     goto setalarmhour
323 decalarmhour:
324     if alarmhour = $00 then fullalarmhour
325     let temp = alarmhour & %00001111
326     if temp = 0 then decalarmhourtens
327     let alarmhour = alarmhour - $01
328     goto setalarmhour
329 decalarmhourtens:
330     let temp = alarmhour & %00110000
331     let alarmhour = temp - $10
332     let alarmhour = alarmhour + $09
333     goto setalarmhour
334 resetalarmhour:
335     let alarmhour = $00
336     goto setalarmhour
337 fullalarmhour:
338     let alarmhour = $23
339     goto setalarmhour
340
341 setalarmmin:
342     if alarmmin > $59 then resetalarmmin
343
344     serout 0,N2400,(254,128,"Alarm Minutes:
",254,192)
345
346     gosub printalarmtime
347
348     pause 100
349     if UP = 1 then incalarmmin
350     if DOWN = 1 then decalarmmin
351     if SET = 1 then setfullweekalarmdisp
352     goto setalarmmin
353
354 incalarmmin:
355     let temp = alarmmin & %00001111
356     if temp = 9 then incalarmmintens
357     let alarmmin = alarmmin + $01
358     goto setalarmmin
359 incalarmmintens:
360     let temp = alarmmin & %01110000
361     let alarmmin = temp + $10
362     goto setalarmmin
363 decalarmmin:
364     if alarmmin = $00 then fullalarmmin

```



## Development - Putting it all together (Programming)

---

```

365     let temp = alarmmin & %00001111
366     if temp = 0 then decalarmmintens
367     let alarmmin = alarmmin - $01
368     goto setalarmmin
369 decalarmmintens:
370     let temp = alarmmin & %01110000
371     let alarmmin = temp - $10
372     let alarmmin = alarmmin + $09
373     goto setalarmmin
374 resetalarmmin:
375     let alarmmin = $00
376     goto setalarmmin
377 fullalarmmin:
378     let alarmmin = $59
379     goto setalarmmin
380
381 setfullweekalarmdisp:
382     serout 0,N2400,(254,128,"Weekend Wakeup:")
383     if fullweekalarm = 0 then disfullweekalarm
384     if fullweekalarm = 1 then enfullweekalarm
385
386 disfullweekalarm:
387     serout 0,N2400,(254,192,"Disabled")
388     goto setfullweekalarmpoll
389 enfullweekalarm:
390     serout 0,N2400,(254,192,"Enabled ")
391     goto setfullweekalarmpoll
392
393 setfullweekalarmpoll:
394     pause 100
395     if fullweekalarm > 1 then resetfullweekalarm
396     if UP = 1 then incfullweekalarm
397     if DOWN = 1 then decfullweekalarm
398     if SET = 1 then setfademins
399     goto setfullweekalarmdisp
400
401 incfullweekalarm:
402     let fullweekalarm = fullweekalarm + 1
403     goto setfullweekalarmdisp
404 decfullweekalarm:
405     if fullweekalarm = 0 then fullfullweekalarm
406     let fullweekalarm = fullweekalarm - 1
407     goto setfullweekalarmdisp
408 resetfullweekalarm:
409     let fullweekalarm = 0
410     goto setfullweekalarmdisp
411 fullfullweekalarm:
412     let fullweekalarm = 1
413     goto setfullweekalarmdisp
414
415 setfademins:
416     if fademins > 60 then resetfademins
417     serout 0,N2400,(254,128,"Fade-in time:
418     ",254,192,#fademins," minute(s) ")
419     pause 100
420     if UP = 1 then incfademins
421     if DOWN = 1 then decfademins
422     if SET = 1 then setdismissmins
423     goto setfademins
424 incfademins:
425     let fademins = fademins + 1

```

## Development - Putting it all together (Programming)

---

```

426     goto setfademins
427 decfademins:
428     if fademins = 0 then fullfademins
429     let fademins = fademins - 1
430     goto setfademins
431 resetfademins:
432     let fademins = 0
433     goto setfademins
434 fullfademins:
435     let fademins = 60
436     goto setfademins
437
438 setdismissmins:
439     if dismissmins > 60 then resetdismissmins
440     serout 0,N2400,(254,128,"Dismiss time:
",254,192,#dismissmins," minute(s) ")
441     pause 100
442     if UP = 1 then incdismissmins
443     if DOWN = 1 then decdismissmins
444     if SET = 1 then save
445     goto setdismissmins
446
447 incdismissmins:
448     let dismissmins = dismissmins + 1
449     goto setdismissmins
450 decdismissmins:
451     if dismissmins = 0 then fulldismissmins
452     let dismissmins = dismissmins - 1
453     goto setdismissmins
454 resetdismissmins:
455     let dismissmins = 0
456     goto setdismissmins
457 fulldismissmins:
458     let dismissmins = 60
459     goto setdismissmins
460
461 save:
462     serout 0,N2400,(254,1,"Saving...")
463     writei2c 0,
(second,mins,hour,day,date,month,year,control,alarmmin,
alarmhour,fullweekalarm,fademins,dismissmins)
464     pause 1000
465     serout 0,N2400,(254,1)
466     goto poll

```

## Development - Putting it all together (Programming)

---

### Setup Explanation

Although there are 466 lines of setup code, which may seem like a lot of programming, many of the blocks are repeated with slight modifications. The setup screen order is as follows:

1. Realtime hour
2. Realtime minutes
3. Realtime seconds
4. Realtime day
5. Realtime date
6. Realtime month
7. Realtime year
8. Alarm hour
9. Alarm minute
10. Weekend wake-up enable/disable
11. Fade-in time (minutes)
12. Dismiss time (minutes)

All the realtime setup screens with the exception of the day screen are displayed as numeric values to the user, and all are stored as binary coded decimals - however since the day values range from 0-7, they are only stored in the least significant nibble of the day byte. All the realtime setup variables, as the names suggest, control the realtime information in the DS1307.

The alarm hour, minute, weekend wake-up, fade-in time, and dismiss-time are all variables which I have created for the alarm clock functions. The alarm hour and minute are compared with their respective realtime values and as such I decided that the best approach would be to store these as binary coded decimals, for two reasons:

1. So that the values can easily be displayed on the LCD (see previous explanation of binary coded decimals for justification).
2. So that the values can easily be compared with the realtime values to check for a wake-up condition when polling.

## Development - Putting it all together (Programming)

---

The weekend wakeup is a simple yes or no value and as such the variable I have assigned to this should be 0 or 1.

The fadeinmins and dismissmins are stored as pure decimals so that they can easily be used in the wake-up block; there is no need for them to be stored as binary coded decimals and infact this would just make things more complicated.

The setup menu system is written such that pressing the set button enters the setup system and on subsequent presses advances to the next setup screen. Because the program I am documenting is the breadboard program, there is no exit button functionality because I hadn't put the necessary button on the breadboard. Pressing the up/down buttons increments/decrements respectively the selected values. For user friendliness I intended to write the setup menu system so that attempting to increment/decrement the values beyond their range causes them to loop back i.e. if the minutes variable is 59 and it is incremented, the value will change to 0, and vice versa.

The main difficulty I faced when coding the setup system was the incrementing and decrementing of the binary coded decimal variables. Simply adding and subtracting a pure decimal value of 1 would not work because this doesn't match the encoding of the variables.

Once I had written the code for this for one screen it was a relatively simple job adapting it for the subsequent screens. Below is the code for one menu screen (the realtime hour setup):

```

4   sethour:                               16
5       if hour > $23 then resethour      17   inhour:
6                                           18       let temp = hour & %00001111
7       serout 0,N2400,(254,128,"Hour:",254,192) 19       if temp = 9 then inhourtens
8                                           20       let hour = hour + $01
9       gosub printtime                   21       goto sethour
10                                          22   inhourtens:
11       pause 100                         23       let temp = hour & %00110000
12       if UP = 1 then inhour             24       let hour = temp + $10
13       if DOWN = 1 then dechour          25       goto sethour
14       if SET = 1 then setmins           26   dechour:
15       goto sethour                       27       if hour = $00 then fullhour

```

## Development - Putting it all together (Programming)

---

```

28     let temp = hour & %00001111
29     if temp = 0 then dechourtens
30     let hour = hour - $01
31     goto sethour
32 dechourtens:
33     let temp = hour & %00110000
34     let hour = temp - $10
35     let hour = hour + $09
36     goto sethour
37 resethour:
38     let hour = $00
39     goto sethour
40 fullhour:
41     let hour = $23
42     goto sethour

```

The first part of the hour setup screen, the 'sethour' block is a sort of miniature polling block; it checks that the hour value is not beyond the valid range, prints the current value to the screen and polls for up/down/set button presses. I have made use of the printtime subroutine that I wrote to increase memory efficiency.

The remaining code blocks all combine to perform the incrementing and decrementing of the binary coded decimal values. Because I couldn't get the program to compile with the enhanced if statements I had to break the logic down into the basic routines as seen above.

In order to correctly increment/decrement the binary coded decimals it was necessary that I take into account the carrying over of units. If the values were stored as pure binary this wouldn't have been necessary, but that was not the case.

The necessary steps for binary coded decimal addition and subtraction are to first of all check the value of the units; if the value is to be incremented and the unit is 9, the unit should be set to 0 and it carried over to the tens; i.e. the tens are incremented. If the units value is not 9 then only the units should be incremented.

If the value is to be decremented and the units value is 0, the units value should be set to 0 and the decrement carried over; the tens value should be decremented. If the value is not 0 then just the units should be decremented.

The above procedure I have described is pretty much primary school addition and subtraction; the challenge was in converting this set of procedures into a working program.

## Development - Putting it all together (Programming)

---

```

17  inhour:
18      let temp = hour & %00001111
19      if temp = 9 then inhourtens
20      let hour = hour + $01
21      goto sethour
22  inhourtens:
23      let temp = hour & %00110000
24      let hour = temp + $10
25      goto sethour
26  dechour:
27  if hour = $00 then fullhour
28      let temp = hour & %00001111
29      if temp = 0 then dechourtens
30      let hour = hour - $01
31      goto sethour
32  dechourtens:
33      let temp = hour & %00110000
34      let hour = temp - $10
35      let hour = hour + $09
36      goto sethour
37  resethour:
38      let hour = $00
39      goto sethour
40  fullhour:
41      let hour = $23
42      goto sethour

```

In addition to the incrementing/decrementing of the values, checks were necessary to ensure that when the values go outside the range, they loop back to the other end of the range. The line that checks if the value has exceeded the maximum has already been described, and the procedure that it jumps to if true can be seen in the above code (the fullhour block).

After my explanation of what the code actually does, it should be reasonably obvious how it works without further description. Because of the problem I had with more advanced IF syntax, it was necessary to describe the logic in a sequence of more basic if statements with goto statements if true.

I adapted this code for use with the realtime minutes, realtime seconds, realtime date, realtime month, realtime year, alarm hour and alarm minute screens, changing of course the maximum values, the screen labels and the printing subroutines used for printing the values to screen.

The day setup screen, weekend wakeup screen, fade-in time, and dismiss time screens used different code because they do not require the procedures for incrementing/decrementing of binary coded decimals.

## Development - Putting it all together (Programming)

---

The day and weekend wakeup screens do not control numeric values; or more specifically, they don't from the users point of view (for maximum ease of use). In the program they are of course controlled by decimals stored in the byte variables; for day ranges from 1 to 7, where 1 is Sunday, 2 is Monday, etc.

For the weekend wakeup variable only two values are necessary; 0 for disabled and 1 for enabled (where 1 will cause the wake-up call to be run at weekends).

As I said these setup screens control non-numeric values from the user point of view yet programmatically they are numeric values. Therefore it was necessary for me to perform a conversion from the numeric value to a text value when the values are displayed on screen, for user convenience. A sample of the code that performs this is below:

```

124 setdayclear:
125     serout 0,N2400,(254,1)
126
127 setdaydisp:
128     serout 0,N2400,(254,128,"Day:")
129     if day = $01 then sunday
130     if day = $02 then monday
131     if day = $03 then tuesday
132     if day = $04 then wednesday
133     if day = $05 then thursday
134     if day = $06 then friday
135     if day = $07 then saturday
136
137 sunday:
138     serout 0,N2400,(254,192,"Sunday ")
139     goto setdaypoll
140 monday:
141     serout 0,N2400,(254,192,"Monday ")
142     goto setdaypoll
143 tuesday:
144     serout 0,N2400,(254,192,"Tuesday ")
145     goto setdaypoll
146 wednesday:
147     serout 0,N2400,(254,192,"Wednesday")
148     goto setdaypoll
149 thursday:
150     serout 0,N2400,(254,192,"Thursday ")
151     goto setdaypoll
152 friday:
153     serout 0,N2400,(254,192,"Friday ")
154     goto setdaypoll
155 saturday:
156     serout 0,N2400,(254,192,"Saturday ")
157     goto setdaypoll
158
159 setdaypoll:
160     pause 100
161     if day > 7 then resetday

```



## Development - Putting it all together (Programming)

---

```

162     if day < 1 then fullday
163     if UP = 1 then incday
164     if DOWN = 1 then decday
165     if SET = 1 then setdate
166     goto setdaydisp
167
168 incday:
169     let day = day + 1
170     goto setdaydisp
171 decday:
172     let day = day - 1
173     goto setdaydisp
174 resetday:
175     let day = 1
176     goto setdaydisp
177 fullday:
178     let day = 7
179     goto setdaydisp

```

The key part of the numeric to text conversion is the setdaydisp procedure, which is a series of IF statements that combine to form a switch statement, directing the program to the appropriate block based on the day value. I used a similar approach for the weekend wakeup enable/disable screen, though with of course only two states; enabled and disabled.

I have included 100ms pauses within all the screen loops, to limit the rate at which the values increment/decrement, and to prevent the user accidentally moving through more than one setup screen at a time.

This concludes the explanation of the breadboard program components. Included on the following pages is the entire breadboard program, in one continuous piece.

## Development - Final Breadboard Program

---

```

1  init:
2      ' Let the LCD Firmware initialise and clear the
screen
3      pause 500
4      serout 0,N2400,(254,1)
5      pause 30
6
7      ' Set up the i2c bus
8      i2cslave %11010000, i2cslow, i2cbyte
9
10     ' Memory locations
11     symbol seconds = b0
12     symbol mins = b1
13     symbol hour = b2
14     symbol day = b3
15     symbol date = b4
16     symbol month = b5
17     symbol year = b6
18     symbol control = b7
19     symbol alarmmin = b8
20     symbol alarmhour = b9
21     symbol fullweekalarm = b10
22     symbol fademins = b11
23     symbol dismissmins = b12
24     symbol temp = b13
25
26     ' Set up the input symbols
27     symbol UP = pin0
28     symbol DOWN = pin1
29     symbol SET = pin7
30
31  poll:
32      ' Poll for lamp level change
33      gosub lamp
34
35      ' Reads data from clock chip
36      readi2c 0,
37      (seconds,mins,hour,day,date,month,year,control,alarmmin,
alarmhour,fullweekalarm,fademins,dismissmins)
38      ' Ensure control value is correct (not user
configurable)
39      let control = 16
40
41      ' Poll for setup button press
42      if SET = 1 then setup
43
44      ' Is it the weekend?
45      'IF b3=7 OR b3=1 THEN
46      '    IF b10 = 1 THEN
47      '      IF b1 = b8 AND b2 = b9 THEN goto
wakeup
48      '    ENDIF
49      'ELSE
50      '    IF b1 = b8 AND b2 = b9 THEN goto wakeup
51      'ENDIF
52
53      if day = 1 then ifallweek
54      if day = 7 then ifallweek
55
56  iftime:
57      if hour != alarmhour then clock

```

## Development - Final Breadboard Program

---

```

58     if mins != alarmmin then clock
59     goto wakeup
60
61  ifallweek:
62     if fullweekalarm = 1 then iftime
63
64  clock:
65     serout 0,N2400,(254,192)
66     gosub printdate
67     serout 0,N2400,(254,128)
68     gosub printtime
69     goto poll
70
71  printtime:
72     let temp = hour & %00110000 / 16
73     serout 0,N2400,(#temp)
74     let temp = hour & %00001111
75     serout 0,N2400,(#temp,":")
76
77     let temp = mins & %01110000 / 16
78     serout 0,N2400,(#temp)
79     let temp = mins & %00001111
80     serout 0,N2400,(#temp,":")
81
82     let temp = seconds & %01110000 / 16
83     serout 0,N2400,(#temp)
84     let temp = seconds & %00001111
85     serout 0,N2400,(#temp)
86     return
87
88  printdate:
89     let temp = date & %00110000 / 16
90     serout 0,N2400,(#temp)
91     let temp = date & %00001111
92     serout 0,N2400,(#temp,"/")
93
94     let temp = month & %00010000 / 16
95     serout 0,N2400,(#temp)
96     let temp = month & %00001111
97     serout 0,N2400,(#temp,"/")
98
99     let temp = year & %11110000 / 16
100    serout 0,N2400,("20",#temp)
101    let temp = year & %00001111
102    serout 0,N2400,(#temp," ")
103    return
104
105  printalarmtime:
106     let temp = alarmhour & %00110000 / 16
107     serout 0,N2400,(#temp)
108     let temp = alarmhour & %00001111
109     serout 0,N2400,(#temp,":")
110
111     let temp = alarmmin & %01110000 / 16
112     serout 0,N2400,(#temp)
113     let temp = alarmmin & %00001111
114     serout 0,N2400,(#temp," ")
115     return
116
117  setup:
118     serout 0,N2400,(254,1)
119

```

## Development - Final Breadboard Program

---

```

120 sethour:
121     if hour > $23 then resethour
122
123     serout 0,N2400,(254,128,"Hour:",254,192)
124
125     gosub printtime
126
127     pause 100
128     if UP = 1 then inchour
129     if DOWN = 1 then dechour
130     if SET = 1 then setmins
131     goto sethour
132
133 inchour:
134     let temp = hour & %00001111
135     if temp = 9 then inchourtens
136     let hour = hour + $01
137     goto sethour
138 inchourtens:
139     let temp = hour & %00110000
140     let hour = temp + $10
141     goto sethour
142 dechour:
143     if hour = $00 then fullhour
144     let temp = hour & %00001111
145     if temp = 0 then dechourtens
146     let hour = hour - $01
147     goto sethour
148 dechourtens:
149     let temp = hour & %00110000
150     let hour = temp - $10
151
152     let hour = hour + $09
153     goto sethour
154
155 resethour:
156     let hour = $00
157     goto sethour
158
159 fullhour:
160     let hour = $23
161     goto sethour
162
163 setmins:
164     if mins > $59 then resetmins
165
166     serout 0,N2400,(254,128,"Minutes:",254,192)
167
168     gosub printtime
169
170     pause 100
171     if UP = 1 then incmins
172     if DOWN = 1 then decmins
173     if SET = 1 then setseconds
174     goto setmins
175
176 incmins:
177     let temp = mins & %00001111
178     if temp = 9 then incminstens
179     let mins = mins + $01
180     goto setmins
181
182 incminstens:
183     let temp = mins & %01110000
184     let mins = temp + $10
185     goto setmins

```

## Development - Final Breadboard Program

---

```

182 decmins:
183     if mins = $00 then fullmins
184     let temp = mins & %00001111
185     if temp = 0 then decminstens
186     let mins = mins - $01
187     goto setmins
188 decminstens:
189     let temp = mins & %01110000
190     let mins = temp - $10
191     let mins = mins + $09
192     goto setmins
193 resetmins:
194     let mins = $00
195     goto setmins
196 fullmins:
197     let mins = $59
198     goto setmins
199
200 setseconds:
201     if seconds > $59 then resetseconds
202
203     serout 0,N2400,(254,128,"Seconds:",254,192)
204
205     gosub printtime
206
207     pause 100
208     if UP = 1 then incseconds
209     if DOWN = 1 then decseconds
210     if SET = 1 then setdayclear
211     goto setseconds
212
213 incseconds:
214     let temp = seconds & %00001111
215     if temp = 9 then incsecondstens
216     let seconds = seconds + $01
217     goto setseconds
218 incsecondstens:
219     let temp = seconds & %01110000
220     let seconds = temp + $10
221     goto setseconds
222 decseconds:
223     if seconds = $00 then fullseconds
224     let temp = seconds & %00001111
225     if temp = 0 then decsecondstens
226     let seconds = seconds - $01
227     goto setseconds
228 decsecondstens:
229     let temp = seconds & %01110000
230     let seconds = temp - $10
231     let seconds = seconds + $09
232     goto setseconds
233 resetseconds:
234     let seconds = $00
235     goto setseconds
236 fullseconds:
237     let seconds = $59
238     goto setseconds
239
240 setdayclear:
241     serout 0,N2400,(254,1)
242
243 setdaydisp:

```

## Development - Final Breadboard Program

---

```

244   serout 0,N2400,(254,128,"Day:")
245   if day = $01 then sunday
246   if day = $02 then monday
247   if day = $03 then tuesday
248   if day = $04 then wednesday
249   if day = $05 then thursday
250   if day = $06 then friday
251   if day = $07 then saturday
252
253  sunday:
254     serout 0,N2400,(254,192,"Sunday  ")
255     goto setdaypoll
256  monday:
257     serout 0,N2400,(254,192,"Monday  ")
258     goto setdaypoll
259  tuesday:
260     serout 0,N2400,(254,192,"Tuesday  ")
261     goto setdaypoll
262  wednesday:
263     serout 0,N2400,(254,192,"Wednesday")
264     goto setdaypoll
265  thursday:
266     serout 0,N2400,(254,192,"Thursday ")
267     goto setdaypoll
268  friday:
269     serout 0,N2400,(254,192,"Friday  ")
270     goto setdaypoll
271  saturday:
272     serout 0,N2400,(254,192,"Saturday ")
273     goto setdaypoll
274
275  setdaypoll:
276     pause 100
277     if day > 7 then resetday
278     if day < 1 then fullday
279     if UP = 1 then incday
280     if DOWN = 1 then decday
281     if SET = 1 then setdate
282     goto setdaydisp
283
284  incday:
285     let day = day + 1
286     goto setdaydisp
287  decday:
288     let day = day - 1
289     goto setdaydisp
290  resetday:
291     let day = 1
292     goto setdaydisp
293  fullday:
294     let day = 7
295     goto setdaydisp
296
297  setdate:
298     if date > $31 then resetdate
299
300     serout 0,N2400,(254,128,"Date:   ",254,192)
301
302     gosub printdate
303
304     pause 100
305     if UP = 1 then incdate

```

## Development - Final Breadboard Program

---

```

306     if DOWN = 1 then decdate
307     if SET = 1 then setmonth
308     goto setdate
309
310 incdate:
311     let temp = date & %00001111
312     if temp = 9 then incdatetens
313     let date = date + $01
314     goto setdate
315 incdatetens:
316     let temp = date & %00110000
317     let date = temp + $10
318     goto setdate
319 decdate:
320     if date = $01 then fulldate
321     let temp = date & %00001111
322     if temp = 0 then decdatetens
323     let date = date - $01
324     goto setdate
325 decdatetens:
326     let temp = date & %00110000
327     let date = temp - $10
328     let date = date + $09
329     goto setdate
330 resetdate:
331     let date = $01
332     goto setdate
333 fulldate:
334     let date = $31
335     goto setdate
336
337 setmonth:
338     if month > $12 then resetmonth
339
340     serout 0,N2400,(254,128,"Month:  ",254,192)
341
342     gosub printdate
343
344     pause 100
345     if UP = 1 then incmonth
346     if DOWN = 1 then decmonth
347     if SET = 1 then setyear
348     goto setmonth
349
350 incmonth:
351     let temp = month & %00001111
352     if temp = 9 then incmonthtens
353     let month = month + $01
354     goto setmonth
355 incmonthtens:
356     let temp = month & %00010000
357     let month = temp + $10
358     goto setmonth
359 decmonth:
360     if month = $01 then fullmonth
361     let temp = month & %00001111
362     if temp = 0 then decmonthtens
363     let month = month - $01
364     goto setmonth
365 decmonthtens:
366     let temp = month & %00010000
367     let month = temp - $10

```



## Development - Final Breadboard Program

---

```

368     let month = month + $09
369     goto setmonth
370 resetmonth:
371     let month = $01
372     goto setmonth
373 fullmonth:
374     let month = $12
375     goto setmonth
376
377 setyear:
378     if year > $99 then resetyear
379
380     serout 0,N2400,(254,128,"Year:  ",254,192)
381
382     gosub printdate
383
384     pause 100
385     if UP = 1 then incyear
386     if DOWN = 1 then decyear
387     if SET = 1 then setalarmhour
388     goto setyear
389
390 incyear:
391     let temp = year & %00001111
392     if temp = 9 then incyeartens
393     let year = year + $01
394     goto setyear
395 incyeartens:
396     let temp = year & %11110000
397     let year = temp + $10
398     goto setyear
399 decyear:
400     if year = $00 then fullyear
401     let temp = year & %00001111
402     if temp = 0 then decyeartens
403     let year = year - $01
404     goto setyear
405 decyeartens:
406     let temp = year & %11110000
407     let year = temp - $10
408     let year = year + $09
409     goto setyear
410 resetyear:
411     let year = $00
412     goto setyear
413 fullyear:
414     let year = $99
415     goto setyear
416
417 setalarmhour:
418     if alarmhour > $23 then resetalarmhour
419
420     serout 0,N2400,(254,128,"Alarm Hour:
",254,192)
421
422     gosub printalarmtime
423
424     pause 100
425     if UP = 1 then incalarmhour
426     if DOWN = 1 then decalarmhour
427     if SET = 1 then setalarmmin
428     goto setalarmhour

```

## Development - Final Breadboard Program

---

```

429
430 incalarmhour:
431     let temp = alarmhour & %00001111
432     if temp = 9 then incalarmhourtens
433     let alarmhour = alarmhour + $01
434     goto setalarmhour
435 incalarmhourtens:
436     let temp = alarmhour & %00110000
437     let alarmhour = temp + $10
438     goto setalarmhour
439 decalarmhour:
440     if alarmhour = $00 then fullalarmhour
441     let temp = alarmhour & %00001111
442     if temp = 0 then decalarmhourtens
443     let alarmhour = alarmhour - $01
444     goto setalarmhour
445 decalarmhourtens:
446     let temp = alarmhour & %00110000
447     let alarmhour = temp - $10
448     let alarmhour = alarmhour + $09
449     goto setalarmhour
450 resetalarmhour:
451     let alarmhour = $00
452     goto setalarmhour
453 fullalarmhour:
454     let alarmhour = $23
455     goto setalarmhour
456
457 setalarmmin:
458     if alarmmin > $59 then resetalarmmin
459
460     serout 0,N2400,(254,128,"Alarm Minutes:
461     ",254,192)
462     gosub printalarmtime
463
464     pause 100
465     if UP = 1 then incalarmmin
466     if DOWN = 1 then decalarmmin
467     if SET = 1 then setfullweekalarmdisp
468     goto setalarmmin
469
470 incalarmmin:
471     let temp = alarmmin & %00001111
472     if temp = 9 then incalarmmintens
473     let alarmmin = alarmmin + $01
474     goto setalarmmin
475 incalarmmintens:
476     let temp = alarmmin & %01110000
477     let alarmmin = temp + $10
478     goto setalarmmin
479 decalarmmin:
480     if alarmmin = $00 then fullalarmmin
481     let temp = alarmmin & %00001111
482     if temp = 0 then decalarmmintens
483     let alarmmin = alarmmin - $01
484     goto setalarmmin
485 decalarmmintens:
486     let temp = alarmmin & %01110000
487     let alarmmin = temp - $10
488     let alarmmin = alarmmin + $09
489     goto setalarmmin

```

## Development - Final Breadboard Program

---

```

490 resetalarmmin:
491     let alarmmin = $00
492     goto setalarmmin
493 fullalarmmin:
494     let alarmmin = $59
495     goto setalarmmin
496
497 setfullweekalarmdisp:
498     serout 0,N2400,(254,128,"Weekend Wakeup:")
499     if fullweekalarm = 0 then disfullweekalarm
500     if fullweekalarm = 1 then enfullweekalarm
501
502 disfullweekalarm:
503     serout 0,N2400,(254,192,"Disabled")
504     goto setfullweekalarmpoll
505 enfullweekalarm:
506     serout 0,N2400,(254,192,"Enabled ")
507     goto setfullweekalarmpoll
508
509 setfullweekalarmpoll:
510     pause 100
511     if fullweekalarm > 1 then resetfullweekalarm
512     if UP = 1 then incfullweekalarm
513     if DOWN = 1 then decfullweekalarm
514     if SET = 1 then setfademins
515     goto setfullweekalarmdisp
516
517 incfullweekalarm:
518     let fullweekalarm = fullweekalarm + 1
519     goto setfullweekalarmdisp
520 decfullweekalarm:
521     if fullweekalarm = 0 then fullfullweekalarm
522     let fullweekalarm = fullweekalarm - 1
523     goto setfullweekalarmdisp
524 resetfullweekalarm:
525     let fullweekalarm = 0
526     goto setfullweekalarmdisp
527 fullfullweekalarm:
528     let fullweekalarm = 1
529     goto setfullweekalarmdisp
530
531 setfademins:
532     if fademins > 60 then resetfademins
533     serout 0,N2400,(254,128,"Fade-in time:
534     ",254,192,#fademins," minute(s) ")
535     pause 100
536     if UP = 1 then incfademins
537     if DOWN = 1 then decfademins
538     if SET = 1 then setdismissmins
539     goto setfademins
540
541 incfademins:
542     let fademins = fademins + 1
543     goto setfademins
544 decfademins:
545     if fademins = 0 then fullfademins
546     let fademins = fademins - 1
547     goto setfademins
548
549 resetfademins:
550     let fademins = 0
551     goto setfademins
552
553 fullfademins:

```

## Development - Final Breadboard Program

---

```

551     let fademins = 60
552     goto setfademins
553
554 setdismissmins:
555     if dismissmins > 60 then resetdismissmins
556     serout 0,N2400,(254,128,"Dismiss time:
",254,192,#dismissmins," minute(s) ")
557     pause 100
558     if UP = 1 then incdismissmins
559     if DOWN = 1 then decddismissmins
560     if SET = 1 then save
561     goto setdismissmins
562
563 incdismissmins:
564     let dismissmins = dismissmins + 1
565     goto setdismissmins
566 decddismissmins:
567     if dismissmins = 0 then fullddismissmins
568     let dismissmins = dismissmins - 1
569     goto setdismissmins
570 resetdismissmins:
571     let dismissmins = 0
572     goto setdismissmins
573 fullddismissmins:
574     let dismissmins = 60
575     goto setdismissmins
576
577 save:
578     serout 0,N2400,(254,1,"Saving...")
579     writei2c 0,
(seconds,mins,hour,day,date,month,year,control,alarmmin,a
larmhour,fullweekalarm,fademins,dismissmins)
580     pause 1000
581     serout 0,N2400,(254,1)
582     goto poll
583
584 wakeup:
585     serout 0,N2400,(254,128,"Wake-up!")
586
587     let b0 = 0
588     let w1 = fademins * 300
589     for b0 = 0 to 200
590         pwmout 3,49,b0
591         pause w1
592     next b0
593
594     let w1 = dismissmins * 1000
595     pause w1
596     pwmout 3,0,0
597
598     serout 0,N2400,(254,1)
599     goto poll
600
601 lamp:
602     readadc10 2,w0
603     pwmout 3,249,w0
604     return

```

# Research - Electronics - Input (Control) Components

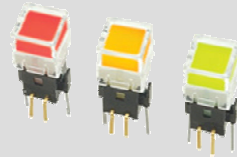
## Square Illuminated PCB Switches

Price: £3.20.

Found at Rapid Online > Electronic Components > Switches > Tactile Switches > Square Illuminated PCB Switches

Illuminated PCB switches with square caps and a soft momentary action. Applications include broadcasting, medical and office automation.

- Contact rating - 12V DC 20mA
- SPDT momentary action
- Rubber contacts
- Red, green or yellow LED
- Minimum 100,000 operations
- 7.62mm x 7.62mm base



Contact resistance 1kΩ maximum  
 Operating force 0.9±0.3N  
 Voltage proof AC 150V 1 minute  
 Soldering 270±5°C 3 sec  
 Insulation resistance 100MΩ minimum  
 DC Operating temp. range -20°C to +70°C  
 Bounce 10msec maximum  
 Storage temp. range -40°C to +80°C  
 Travel 1.0±0.3mm  
 Operating life 100,000 operations

## Requirements

I didn't do any research into control input components before the development stage because I knew that I could only be sure of my requirements after having come up with system ideas and developed the program.

I now know that I will require a minimum of two momentary action digital inputs for the setup menu - one to increase the selected value, and one to change the value selection.

However, for ease of use I think four momentary action digital inputs is a better choice. Two will be used for incrementing and decrementing the selected value, one to move the value selection along, and one to move the value selection back (this will require alteration of the program).

- These square illuminated switches fulfil the requirement of the specification in that they are illuminated, which will make them easy to reach in the dark.
- However they are PCB mounting, which means I would have to put them on PCBs and then attach the PCB to the case front panel.

If I use these switches I would choose the yellow version; red is too harsh a colour and green is already used for the LCD.

# Research - Electronics - Input (Control) Components

## Multi-direction tactile switch large

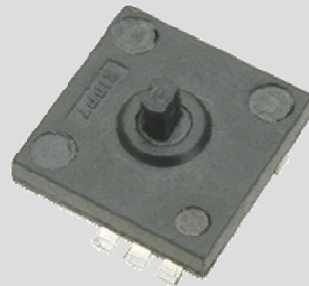
Price: £1.40

Found at Rapid Online > Electronic Components > Switches > Microswitches > Multi-direction tactile switch large

Order code 78-0862

A value for money multi-direction tactile switch that is suitable for use in PDAs, digital cameras, etc.

- Up, down, right, left or push actuation
- Lightweight
- Positive click provides good tactile feedback
- Normally open contacts
- Minimum 1,500,000 operations for extended life



Contact rating 24V DC 50mA  
 Contact resistance (max) 200mΩ  
 Insulation resistance (min) 100MΩ

## Advantages

Using this multi-direction tactile switch would allow me to create a very intuitive menu system.

I believe that by using this system pretty much anyone would be able to work out how to use the alarm clock without any instructions!

I would connect it up such that moving the stick to the right enters setup and advances through the setup pages, while moving the stick to the left would move back through the pages.

When in a setup page, moving the stick up/down would increment/decrement the selected value.

## Disadvantages

For the purpose of good aesthetics I could not use this switch with the stick it has without adding some sort of "D-Pad" (directional pad as found on mobile phones etc.). However I was not able to find one on the Rapid website.

Additionally no illumination is provided so the best that I could do to ensure visibility in the dark would be to surround it by four LEDs, each at the centre of an edge.

# Research - Electronics - Input (Control) Components

## Low profile push switches

Price: £0.66 (for the black push to break - which is what I would use).

Found at Rapid Online > Electronic Components > Switches > Push-Button Switches > Low profile push switches

A range of low profile push switches with large 9mm actuator button.

- Available in momentary action push to make, push to break or changeover options
- Smart chrome bezel
- Supplied with button in red or black
- SCI R13-502 series



Contact rating 3A 125V AC  
1.5A 250V AC  
Contact resistance 50mΩ max.  
Insulation resistance >10 x 10<sup>7</sup>Ω; min.  
Body dimensions push to make 20mm x 14.0mm dia,  
push to break 22.3mm x 14.0mm dia.  
SPDT 21mm x 14.0mm dia.  
Panel cut-out 12.7mm

## Advantages

These switches are panel mounting which would avoid the problem I would face with the others of having to produce PCBs for them! Even having produced PCBs for mounting the others, it would be difficult to mount them such that there is no gap between the switch and the panel cut-out whereas with these there would be none.

They are also very simple in terms of connection; having only two solder tags to connect to means I wouldn't even need to consult data-sheets. Compare with the illuminated switches for example which have four connections for switching and an additional two the illumination.

## Disadvantages

No illumination; again I would have to improvise and mount an LED by each on the front panel to make them easy to find.

The black push to break version's order code is 78-1570.



## Research - Electronics - Input (Control) Components

---

### 16mm soft touch knob

Price: £0.12

Found at Rapid Online > Tools, Fasteners & Production Equipment > Fasteners & Fixings > Knobs > 16mm soft touch knob

Soft touch technology available in a range of 16mm diameter mixer style control knobs. The knobs feature a soft touch matt black body with a choice of seven coloured pointers moulded into the body.

- Styled with todays front panels in mind
- Offers superb looks at competitive costings
- Designed to push-fit onto 6mm splined shaft controls
- Rean P670 series

Available with red, green, blue, white, yellow, blue, white, grey and orange pointers.



### Requirements

The lamp dimmer override potentiometer will need a knob for when it is mounted to the case. The shaft diameter of the potentiometer I used when breadboarding is 6mm, so one of these is ideal. I chose the green version, to match the backlight of the replacement LCD board that I decided I would use. The order code is 32-0455.

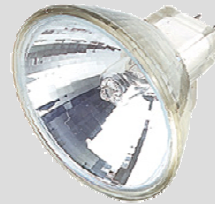
## Research - Electronics - Output (Wake-up Call) Components

### Dichroic low voltage lamps

Price: Varies from £1.20 to £1.59 depending on version.

Found at Rapid Online > Electrical & Power > Elect Prod & Lighting > Lighting > Dichroic low voltage lamps

A range of 12V spot lights which are safe, easy to use and have low power consumption, providing excellent light output which runs cool making them ideal for display work.



- Available in a range of beam angles in open or enclosed (protects the reflector and filament bulbs from finger prints) versions

Operating voltage 12V Average life 4000 hours

Lamp base GUS.3

Style Wattage Beam Ansi code Order code

Open 35W 20° FRA 23-1900

Open 50W 24° EXZ 23-1905

Open 50W 38° EXN 23-1910

Closed 20W 36° - 23-1912

Closed 35W 40° FMW+C 23-1940

Closed 50W 24° EXZ+Z 23-1945

Closed 50W 38° EXN 23-1950

Closed 50W 60° FNV-P 23-1951

### Requirements

For development purposes I had been using a 20w halogen bulb, without any form of reflector/lens etc.

When I convert the results of my breadboarding and programming adventures into an assembled program I will need more than just a bare bulb for the dawn simulation.

Ideally, the light output will be very bright but unfocused so as to reduce the dazzling effect - this is highly undesirable because it's likely to make the user close their eyes.

When I came up with my casing ideas, I had hoped that I would be able to have a very wide angle of light radiation, and "frost" the lens somehow to aid lamp diffusion - perhaps by sand blasting it.

Unfortunately the lamp pictured opposite is pretty much the only lamp I could find that is even half suitable; there are fewer 12V lamps than there are 230V lamps so for this reason the range I could choose from was already limited.

I thought the 20W lamp I used during development was slightly on the dim side and so have decided to use a 50W lamp. Because I want a high radiation angle, I have chosen the 50W 60° enclosed version, order code 23-1951, priced £1.59.

## Research - Electronics - Output (Wake-up Call) Components

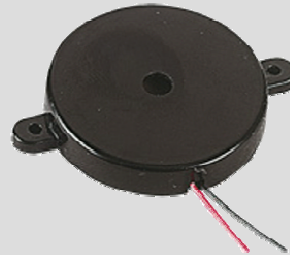
### Miniature piezo transducer

Price: £0.56

Found at Rapid Online > Electrical & Power > PA & Audio / Video > Sirens/Sounders/Transducers > Miniature piezo transducer

A miniature flange mounting piezo audio transducer with flying leads.

- Housed in a compact package
- Requires an external drive circuit
- Soundtech type SEP-1126



Operating voltage (Vp-p max) 30V  
 Sound output 95dB at 10cm  
 Resonant frequency 2.8kHz  
 Current consumption 8mA  
 Capacitance (±30%) 18,000pF  
 Weight 4g

### Requirements

Up until now I had not given any thought to the sound output that will provide the fail-safe wake-up call because it was such a minor issue compared to the other problems.

The main difficulty I faced when finding an appropriate sound output device was getting the tone right. This is of course impossible to judge from a picture on a website.

To ensure the tone is as I think appropriate for an alarm clock, I opted to use a driverless piezo transducer. The reason for this is that the lack of a driver ensures I will have full control over the tone of the audible wake-up call, by using the built in command of the PICAXE for driving piezos.

Now that I had narrowed my choice down to driverless piezos, I had to choose a specific model. This was simply a case of looking at the specifications of every one on the range to see which was loudest. Unfortunately, I couldn't find one as loud as I would have liked, but this was a compromise I was willing to make; I believe the tone is more important than the loudness.

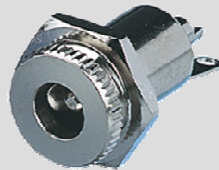
The piezo I chose has an output of 95dB at 10cm, order code 35-0240.

## Research - Electronics - Power Supply Components

### Heavy duty DC power socket

Price: £0.55

Found at Rapid Online > Cables & Connectors > Connectors - Mains/Power > DC Power Connectors > Heavy duty DC power socket



High quality DC power sockets featuring nickel plated bodies and single hole fixing.

- Suitable for panel thickness up to 2.5mm
- Panel cut-out 11mm
- Supplied with hexagonal fixing nut

### Requirements

The alarm clock will need a power supply of minimum capacity 50W since that is the power rating of the lamp I have chosen. The rest of the components are almost negligible; however to ensure maximum reliability, I have allowed an extra 15W for the other components and chosen to use a 65W power supply.

The power supply I have chosen has a 2.5mm jack and thus I chose the 2.5mm version of the heavy duty DC power socket shown opposite, order code 20-1072.

### 30-60w Mini desktop switch mode PSU

Price: £17.75

Found in the Rapid Electronics Catalogue (Paper) 2007 > Electrical & power > Power supplies > page 435

A range of high quality switch mode PSUs that have 3-pin IEC input connectors.  
 Wide range input voltage 90 to 264v AC at 47 to 63Hz  
 Short circuit, over voltage and over current protection  
 Typically 80% efficient  
 Fully regulated output  
 Connector 2.1 x 5.5 x 12mm female barrel  
 Dimensions 120 x 60 x 36mm  
 IEC, TUV, UL approved

## Production - Schematic 1.0

Shown on the right is the schematic for the first artwork I created.

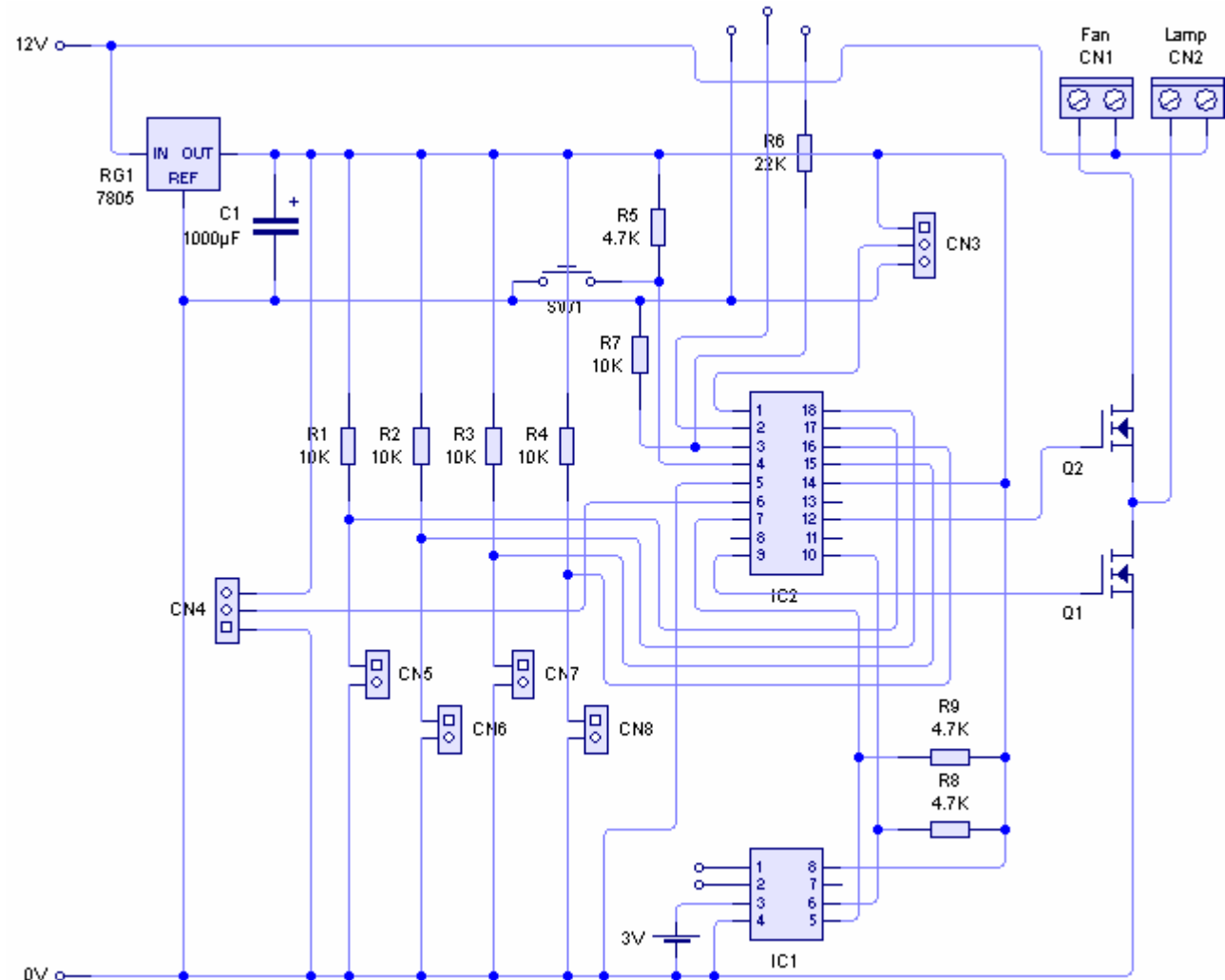
The LCD module is connected at CN4, the switches at CN5 to CN8, the lamp override dimmer at CN3, and the fan and lamp at their labelled terminal blocks.

I have also added a PTM tactile switch, SW1, between the PICAXE reset pin and ov.

The most significant change however is how the fan is driven. It is driven by a separate MOSFET which in turn is powered by the lamp MOSFET. This is because the fan must only come on during the wake-up call and not when the lamp override dimmer is used.

Because the 18X only supports pwmout on one pin (9), and pwmout is needed for the fan too, I have got around this problem by running them from the same PWM output.

The fan is only on however when pin 9 is high too; which will only occur in the program during the wake-up call.



Schematic 1.0

## Production - Artwork 1.0

Shown on the right is the first artwork I designed, which is based on the schematic on the previous page.

I designed this artwork prior to completing all necessary research, in the hope that it would allow me to produce a working PCB sooner.

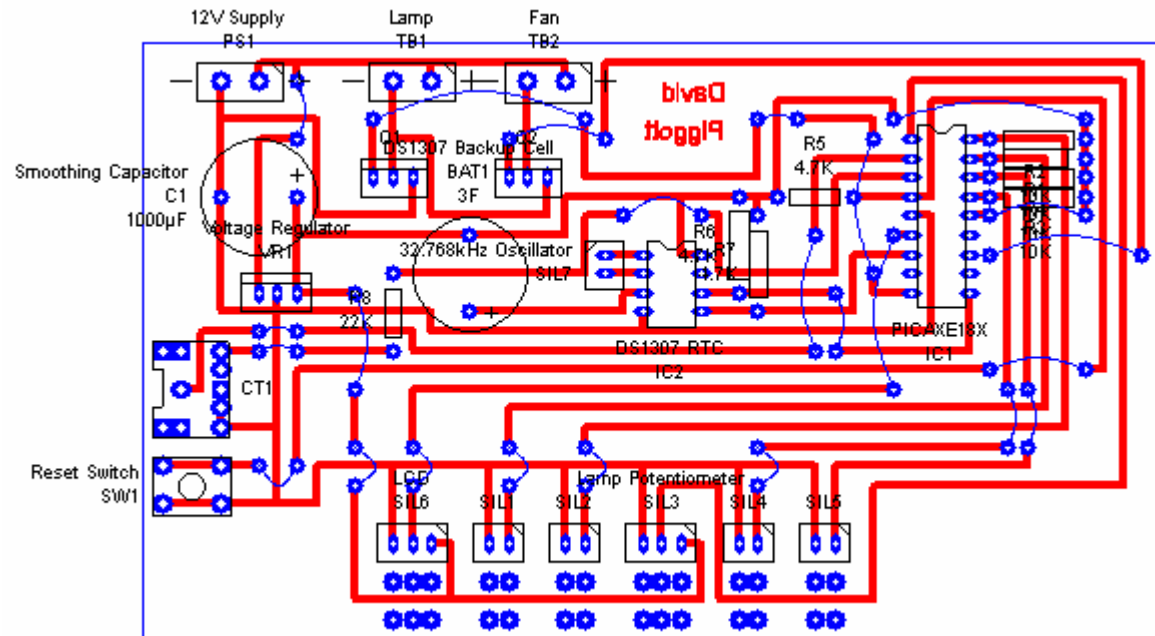
Though I never actually manufactured a PCB from this artwork I have included it for the sake of completeness.

The first is because I have decided to use the square illuminated push to make buttons; this artwork is designed for push to breaks (the difference is that the switch potential dividers are inverted).

The second is that I forgot to add any tracking or pads for the piezo transducer (remember I hadn't decided to use one when I designed this artwork) - the piezo is also missing on the schematic on the previous page.

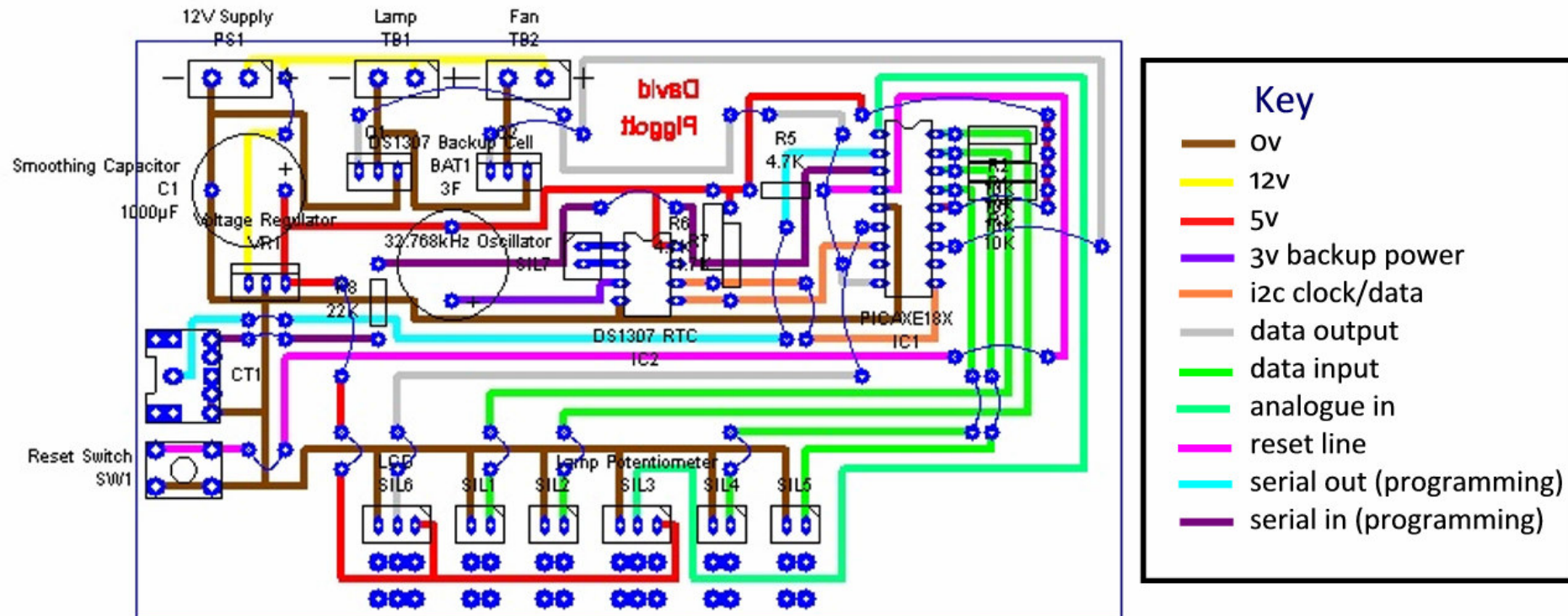
I took special care when designing it to ensure that I did not use wire links for any high current paths - that is, the connections between the source and drains of the MOSFETS. Terminal blocks are used for high current connections to allow the use of thicker wires.

Also worth noting is that I the artwork of a 0.4" capacitor for the positioning of the backup cell holder since PCB Wizard (the software package used to create all my artworks) doesn't have a backup cell holder in it's parts bin. If I had actually made a PCB from this artwork I would have found that the spacing of the pads wasn't sufficient.



Artwork 1.0

## Production - Artwork 1.0 (Colour Coded)



Colour Coded Artwork 1.0

Having suffered large setbacks on previous projects due to making mistakes when I converted my schematics into artworks, I decided to put extra effort into ensuring that there were no mistakes with the tracking of my artworks.

In order to do this, I colour coded the tracks based on what they are for. This made it much easier to check for errors, as you can see above. To colour code them, I copied and pasted the artwork into Microsoft Paint and used the fill tool, manually changing the colour of each area of track. This was necessary because PCB Wizard doesn't support colour coding. I then used Microsoft Photodraw to create the key.



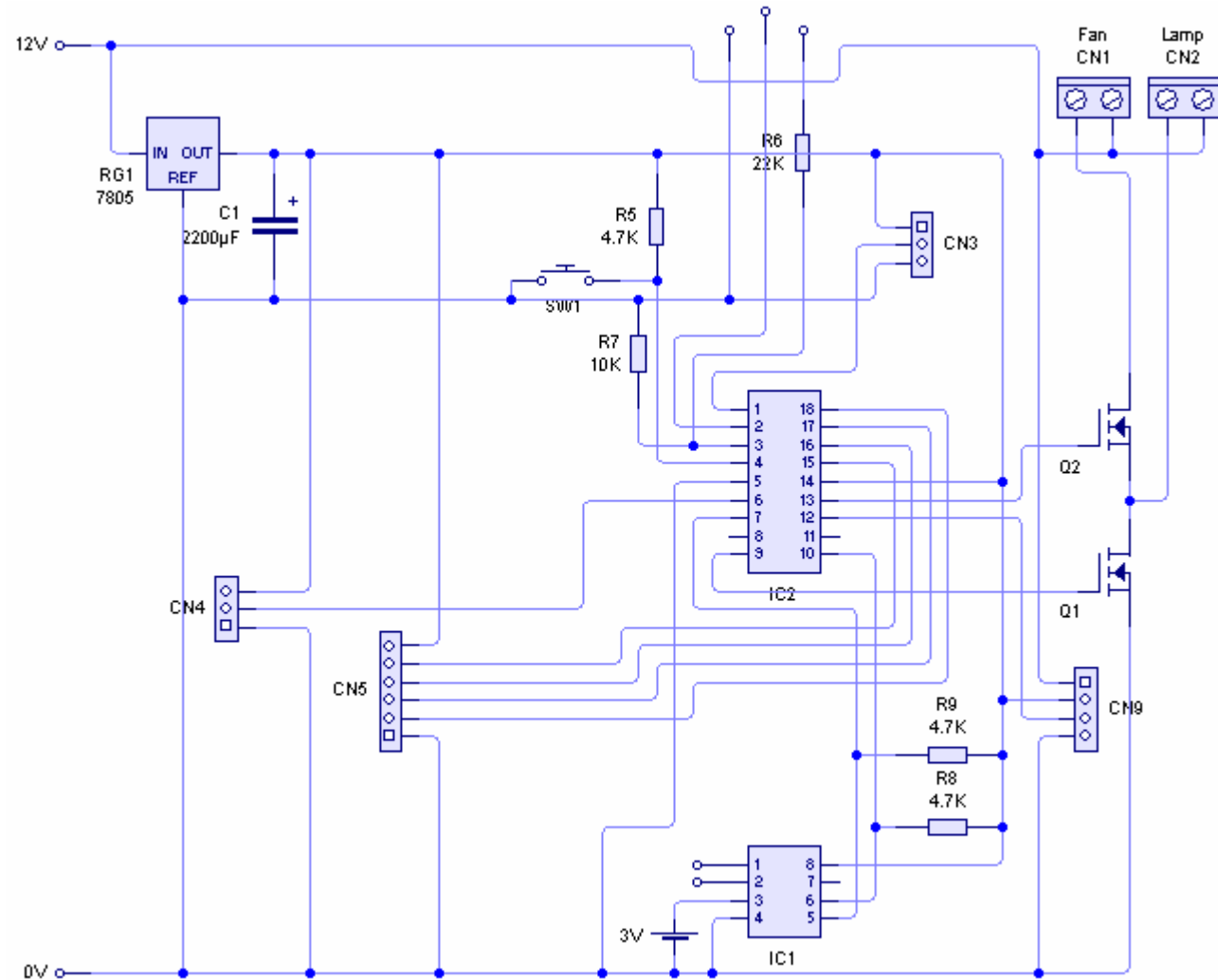
## Production - Schematic 2.0

As explained on the previous pages there were a few flaws in artwork 1.0. As such it was necessary to create a new one. Shown on the right is the revised schematic for this new artwork.

The main change is that the potential dividers for the switches have been removed, and instead the four input pins of the PICAXE are connected to the 6 pin SIL, CN5. Also connected to this SIL are +5v and 0v so that a potential divider can be created on the switchboards, and so that the buttons have power for illumination.

Because at this stage I hadn't actually decided what sort of sounder I would use, I didn't know the requirements of it would be on the circuit.

So that this didn't disrupt my workflow, I decided to cater for every possibility when it came to the sounder requirements. This is why CN9 has +12, +5v, an output pin from the PICAXE, and 0v. This would allow me to use devices with and without drivers, and at different voltages.



Schematic 2.0



## Production - Artwork 2.0

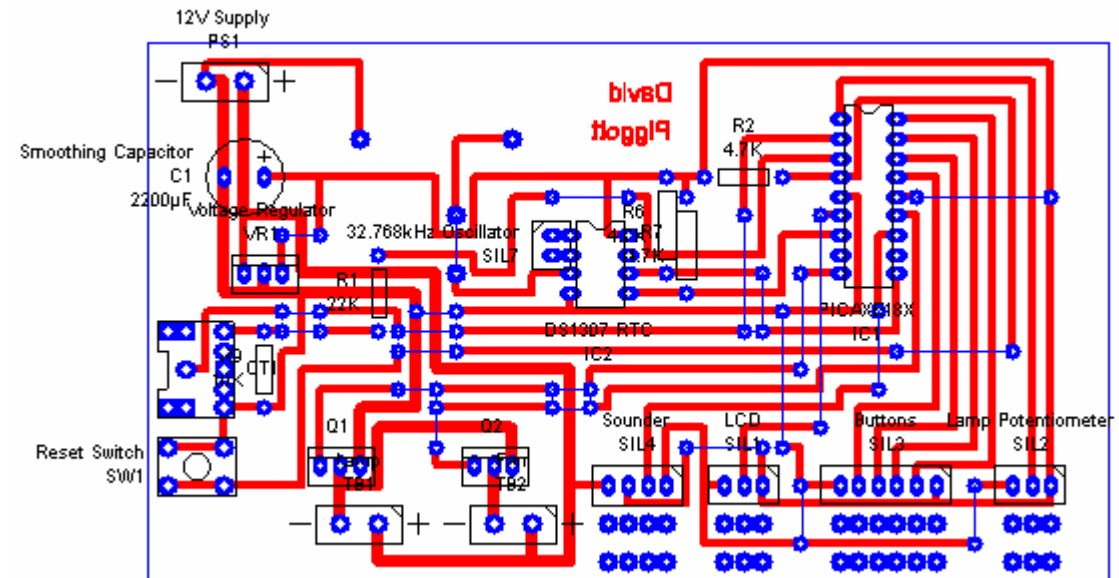
Shown top right is artwork 2.0. The layout is slightly different to that of artwork 1.0. Rather than using the mask for a 0.4" capacitor for the button cell holder, I marked it on just by its solder pads.

In order to get the spacing correct, I held the cell holder up against the computer screen with the zoom level set to 100% - because the screen was an TFT running at native resolution I could be sure that the scaling matched real life.

As in the schematic on the previous page, I have changed the size of the power supply smoothing capacitor. This is because several times during testing, the whole circuit had "crashed". The "crashes" were caused by me bringing the lamp intensity up too quickly with the override dimmer. By increasing the capacity of the smoothing capacitor I hoped to remedy this situation. The larger capacitor was not featured in schematic 1.0 and artwork 1.0 because as already stated, I created them before I had finished development (specifically, before having implemented the lamp dimmer on breadboard).

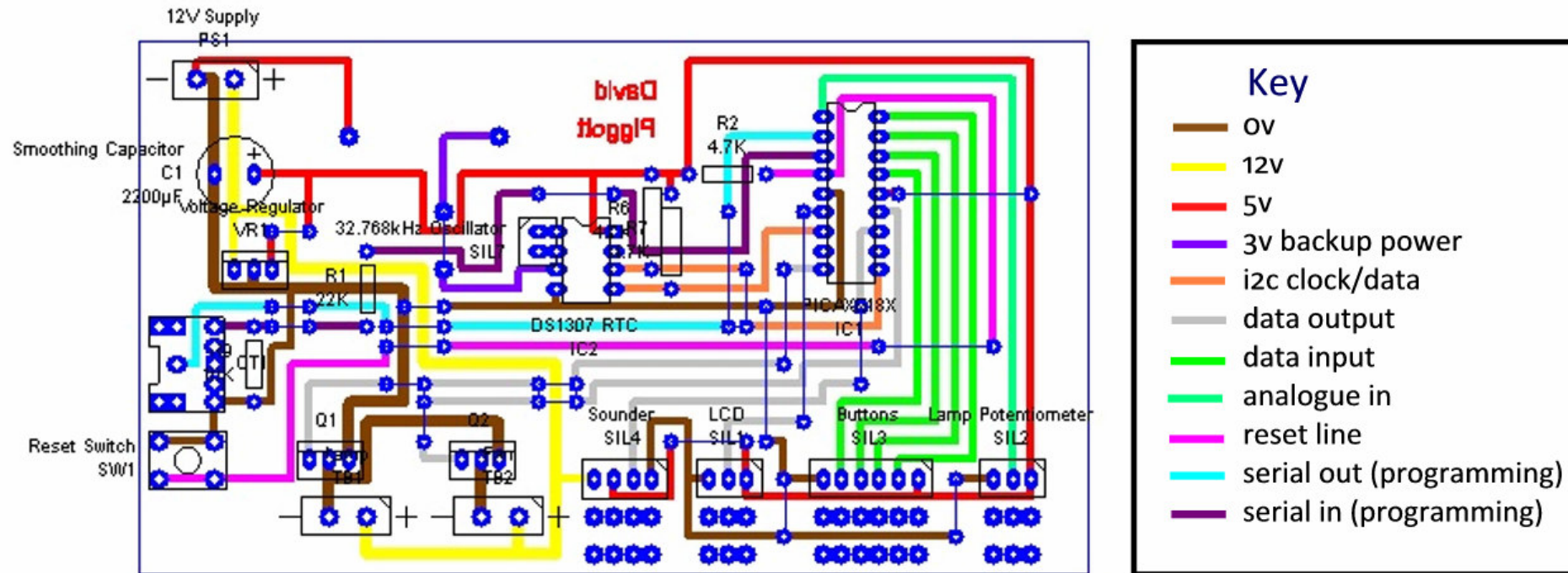
In addition to ensuring that no wire links were used along the source and drain routes to the MOSFETs, I have increased the track thickness to ensure minimum resistance along these high current routes. The lamp and fan MOSFETs and terminal blocks have been relocated to the lower edge of the board so that all connections from front panel components are to the same edge of the PCB. This should simplify the routing of wires within the case.

At this stage I hadn't received the illuminated switches I had ordered and so was unable to create the artwork for the switchboards (there was no datasheet for them).



Artwork 2.0

## Production - Artwork 2.0 (Colour Coded)



Colour Coded Artwork 2.0

As with artwork 1.0, and for the same reasons, I have colour coded artwork 2.0, to help me check the tracking for errors. I used the same technique and software as with artwork 1.0.

## Production - PCB 1.0 from Artwork 2.0

Not connected in the picture are the control switchboards and the piezo sounder. This is because they had only just been delivered. The fan and lamp aren't for reasons described overleaf.

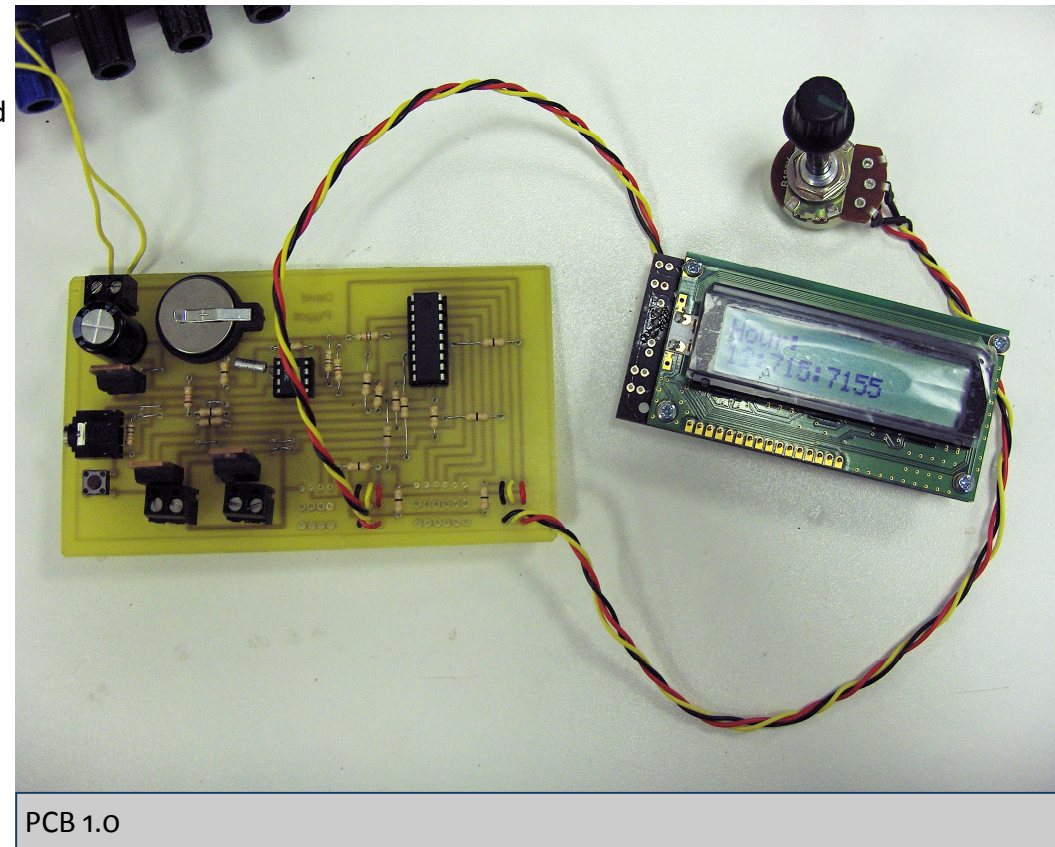
I had however, connected the backlit 16 x 2 LCD display that I ordered (featured in the research I did at the start of this documentation). I gave this priority when connecting things up because I was less sure as to whether the LCD would be compatible.

I faced two problems when connecting the new LCD module. The first was that the datasheet did not state what voltage or series resistor size the backlight needed.

Thus I had to work out what series resistance to use by trial and error; I started with a 1K resistor but it was too dim. Then I tried a 270R resistor - still too dim. So was a 100R resistor. So was a 30R resistor. So was a 22R resistor. In the end, I settled with a 10R resistor.

Because this resistor is connected on the back of the serial driver board, it was necessary to separate the two boards to change the resistor and put them together again to test the new value. This is why in the photo, the SIL that connects them is not soldered onto the LCD board.

The second problem was that I had not tracked on the PCB for the power connections of the LCD - the backlight is unpowered in the picture - I had managed to test it by touching wires directly from the power supply to the backlight power connections on the serial driver board.



## Production - Testing PCB 1.0

---

PCB 1.0 as pictured on the previous page partly worked - some things did and some things didn't. Those that didn't are described on this page.

The first thing noticeable in the picture is that the numbers showing on the LCD are out of range. This wasn't actually a real problem and was caused by the unreliable connection between the LCD board and serial driver board - remember I hadn't soldered them together, so that I could find the correct backlight resistor value - the other reason for not soldering them together was so that I can easily try out different ways of attaching it to the case.

The first problem is as I already described, that I hadn't tracked any power connections on the main PCB for the LCD backlight.

The second and most significant problem is the reason the fan and lamp are not connected in the picture. When I connected the lamp and switched it on using the override dimmer, the MOSFET driving the lamp got quite hot quite quickly. This was presumably because I was now using a 50W bulb instead of the 20W bulb I had used for breadboarding and the current being drawn was too high.

### Summary

To conclude, I needed to do the following:

1. Add connections for the LCD backlight power.
2. Reposition a lot of components to make room for heatsinks to cool the MOSFETs.
3. Work out the pin layout of the illuminated switches and create artworks for the switchboards.

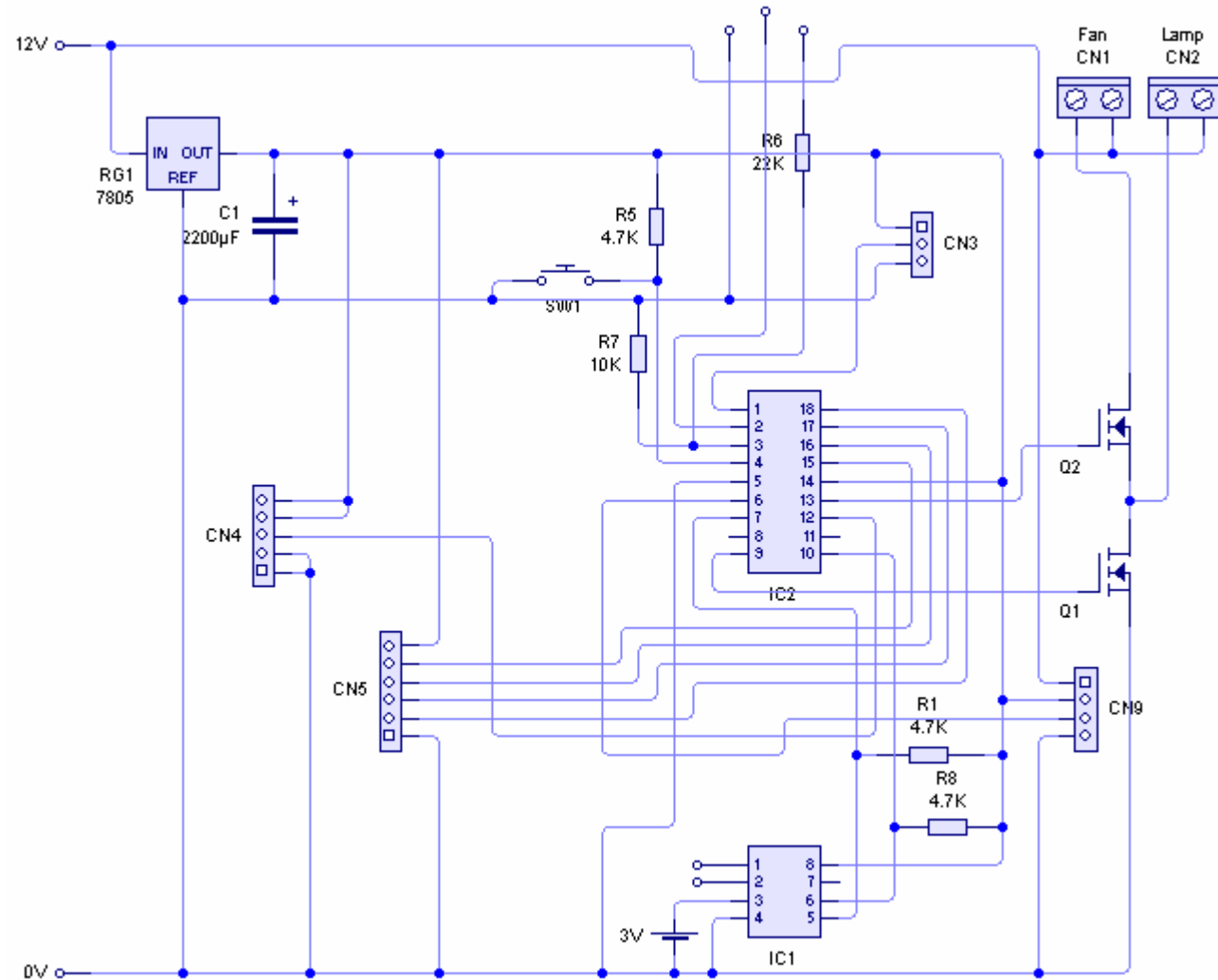
I did not need to research or source the heatsinks because I had a pair of TO-220 heatsinks spare from another project (Micromouse 2005).

## Production - Schematic 3.0

Schematic 3.0 is pictured opposite. It is identical to schematic 2.0 in every way except that CN4 (LCD connector) has been expanded from 3-way to 5-way, so that the LCD backlight power can be connected.

The other change is that the LCD and sound lines from the PICAXE have been swapped. That is, the LCD serial line is now connected to pin 12 and the sounder line to pin 6, whereas before they were the other way around.

This change doesn't improve the circuit or schematic in anyway. The change was for the sake of the artwork - it makes it tidier - and so I updated the schematic to avoid confusion.



Schematic 3.0



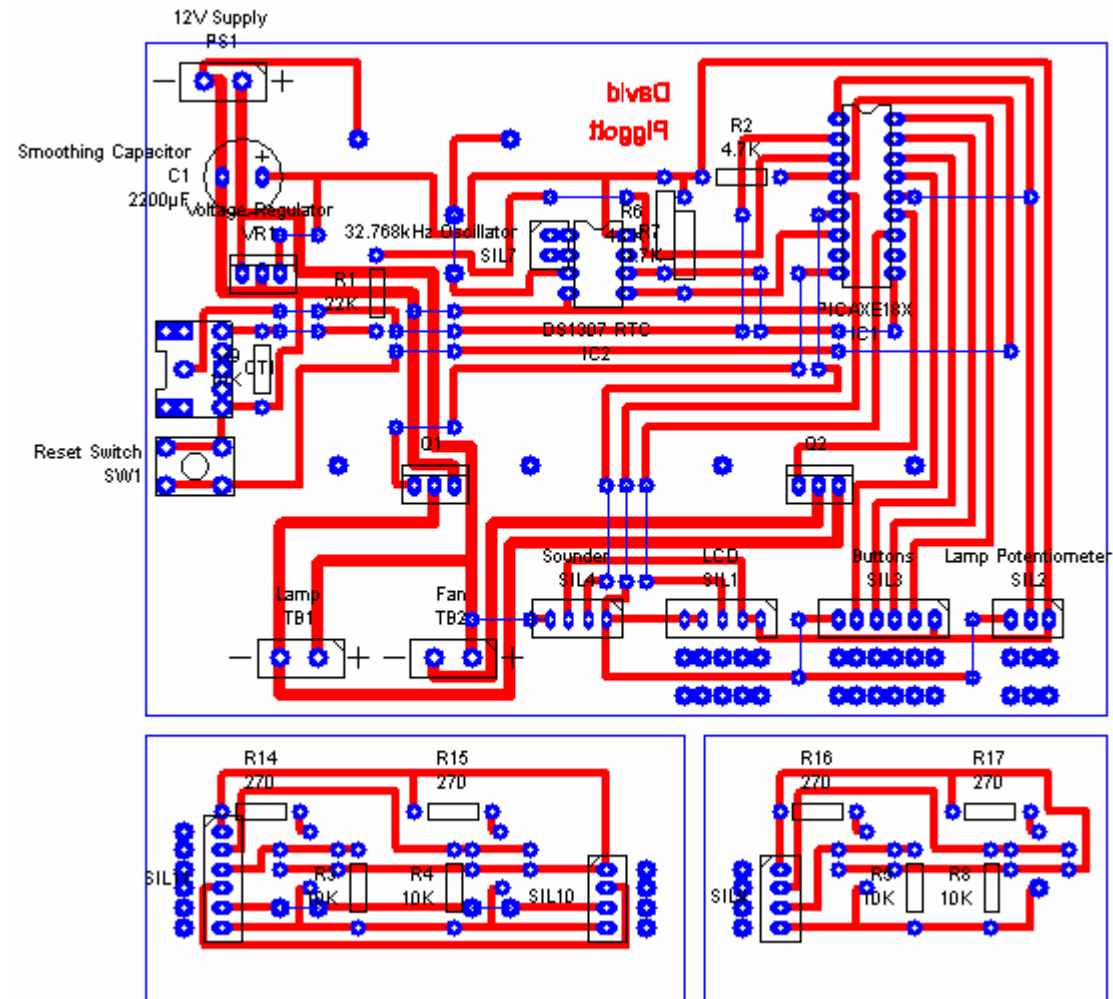
## Production - Artwork 3.0

Artwork 3.0 is pictured opposite. The two changes are the addition of extra power connections on the LCD SIL, and the addition of pads and space for the mounting of heatsinks to the MOSFETs.

As with the button cell holder, PCB Wizard did not have the component mask or pad spacing information for the heatsinks I used, so I used my technique of holding them up the screen with the zoom set to 100% so that I could get the pad spacing correct, and ensure that the heatsinks wouldn't interfere with surrounding components (they are quite big!).

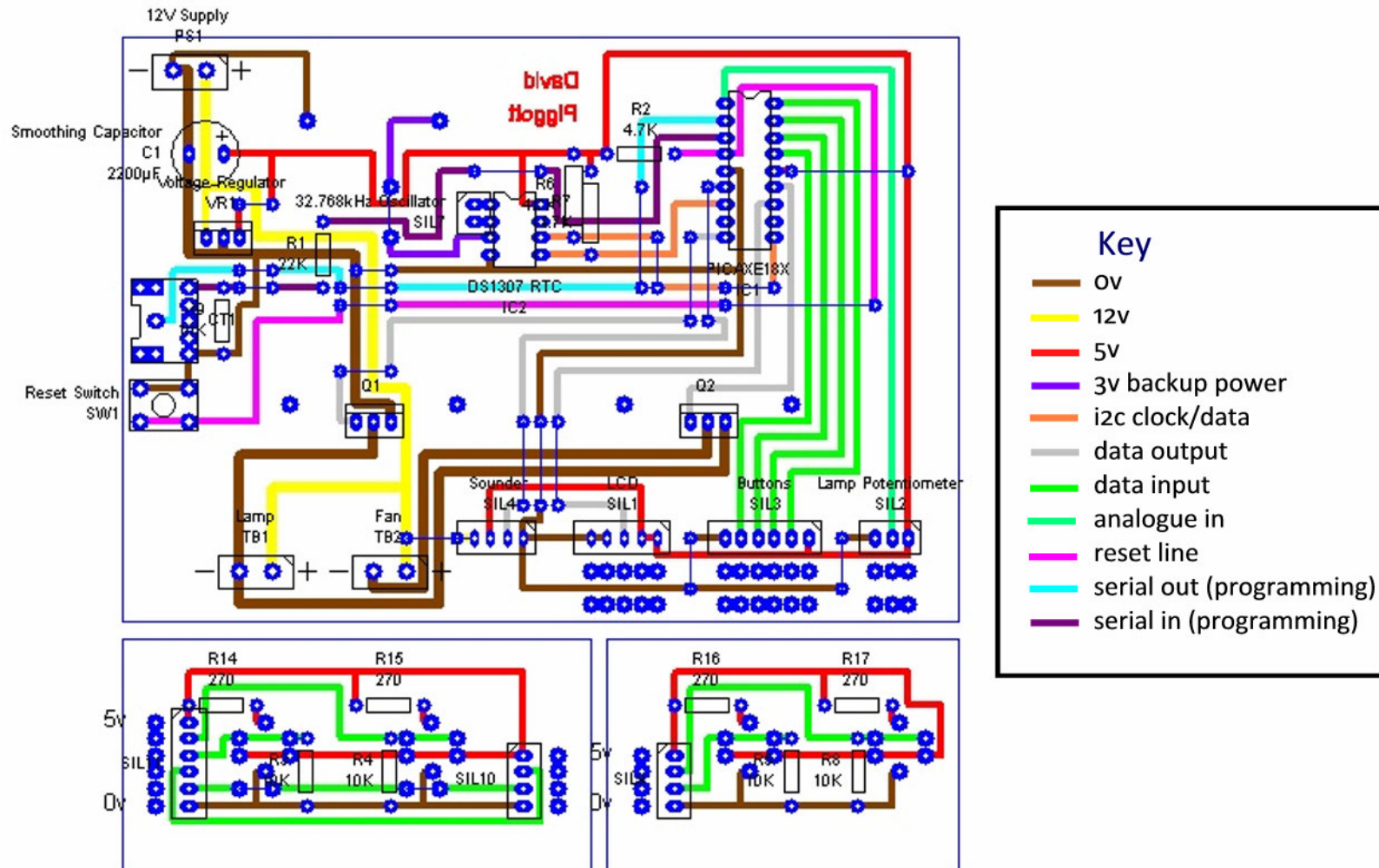
There are an extra two smaller artworks below the main board. These are the switchboards. As already explained, I am using the illuminated PCB mounting switches. Because I intend to position the override dimmer between the up/down and set/exit buttons, it was necessary for me to have these buttons split across two boards, allowing the potentiometer to be mounted between them.

PCB Wizard did not have the component masks or pad spacing information for the illuminated switches, so I had to use my technique of holding them up to the screen at 100% zoom. I had to disable the snap to grid feature too for the power connections.



Artwork 3.0

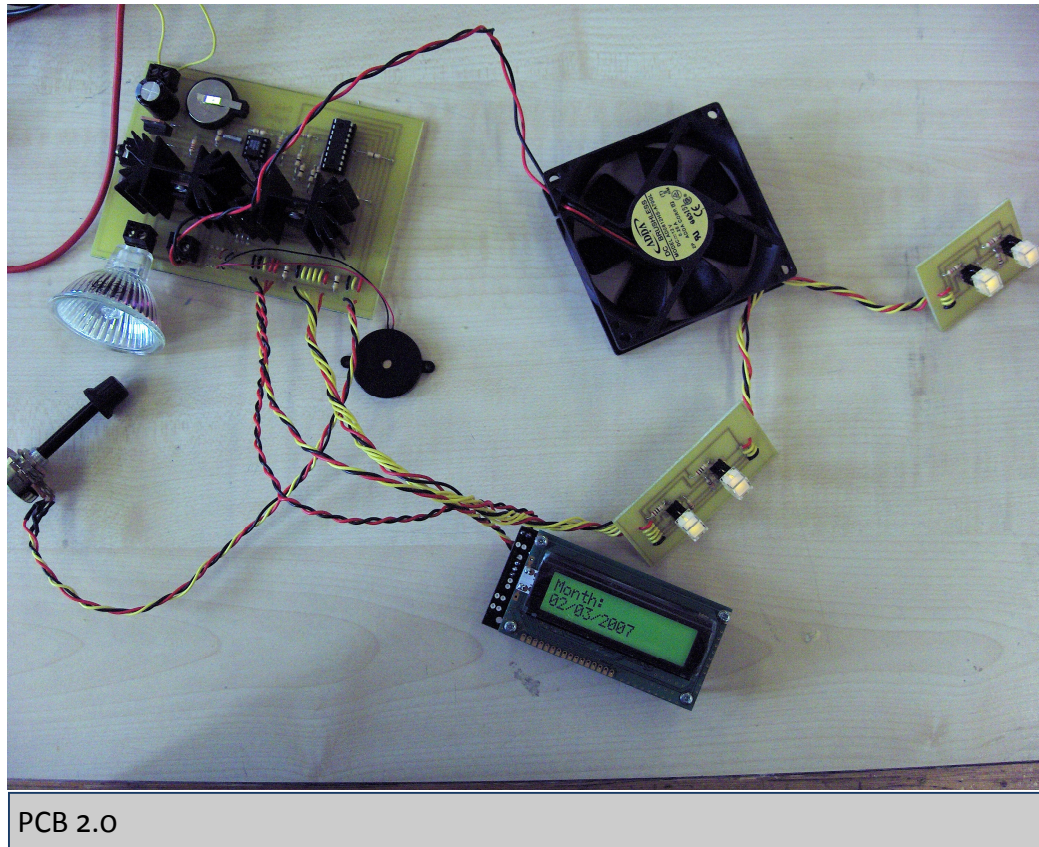
# Production - Artwork 3.0 (Colour Coded)



As with the first artwork, I have colour coded artwork 3.0. This was to help me check for errors in the tracking. I used the same technique and the same software.

Colour Coded Artwork 3.0

## Production - PCB 2.0 from Artwork 3.0



Shown opposite is PCB 2.0 created from artwork 3.0.

Because of the changes to what was connected to what of the PICAXE, I had to modify the program for this PCB. I also had to add to the program the code for the functionality of the piezo sounder. This is explained later in the documentation.

As can be seen in the picture, every component is finally connected!

The buttons, LCD, dimmer potentiometer, and piezo worked flawlessly.

However, I had some trouble with the lamp and fan; not through faults of those components, but because of the MOSFETS.

Initially it all worked as it should have except for the fan which didn't run at all. I tested the MOSFET that should have been driving the fan (by swapping it with the lamp MOSFET) - and the lamp didn't light, indicating that that MOSFET was faulty. To remedy this, I put the original lamp MOSFET back

and ordered a replacement one for the fan.

However, on soldering this replacement in, it caused even more problems; on connecting power to the board, the PICAXE got very warm to the touch very quickly. I removed power and thought about what could be causing this. The only change I had made was soldering the replacement MOSFET in. I then powered it again and it all worked. This explanation is continued overleaf.

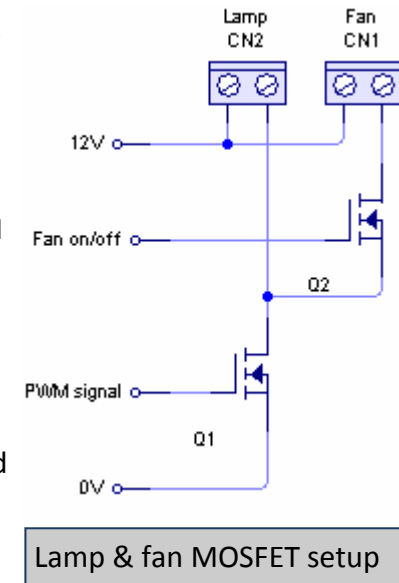


## Production - Testing PCB 2.0

As I have already explained, the fan should not run while the alarm clock is being used as a night light. Therefore it was necessary for me to have separate control of the fan in the program.

However, the fan must have variable intensity for the wake-up call and therefore needs to be run from a pwm capable output; there is only one of these on the 18X. To get around this problem, I came up with the system shown on the right. It should be pretty self explanatory; the lamp is driven by MOSFET Q1 which receives a PWM signal, while the fan is also controlled by MOSFET Q1 but only on when MOSFET Q2 is also switched on by the PICAXE.

I never breadboarded it because I only had only ordered one MOSFET for breadboarding (I hadn't come up with the idea for the lamp override dimmer when I ordered the initial components). Therein lies the problem; this part of the circuit was untested and as it happens doesn't work. My evidence for this is that when I disconnected the fan MOSFET from the lamp MOSFET which drives it, the circuit functioned as expected (except for the fact of course that the fan couldn't run).



The cause of the problem is that when the PWM MOSFET is off, there is infinite resistance between the source of the fan MOSFET and ground. I suspect that this was subsequently causing a voltage of 12V at the gate pin of the fan MOSFET and thus on the output pin of the PICAXE that drives it. This would explain why the PICAXE overheated as it did.

In order to fix this, I had two options:

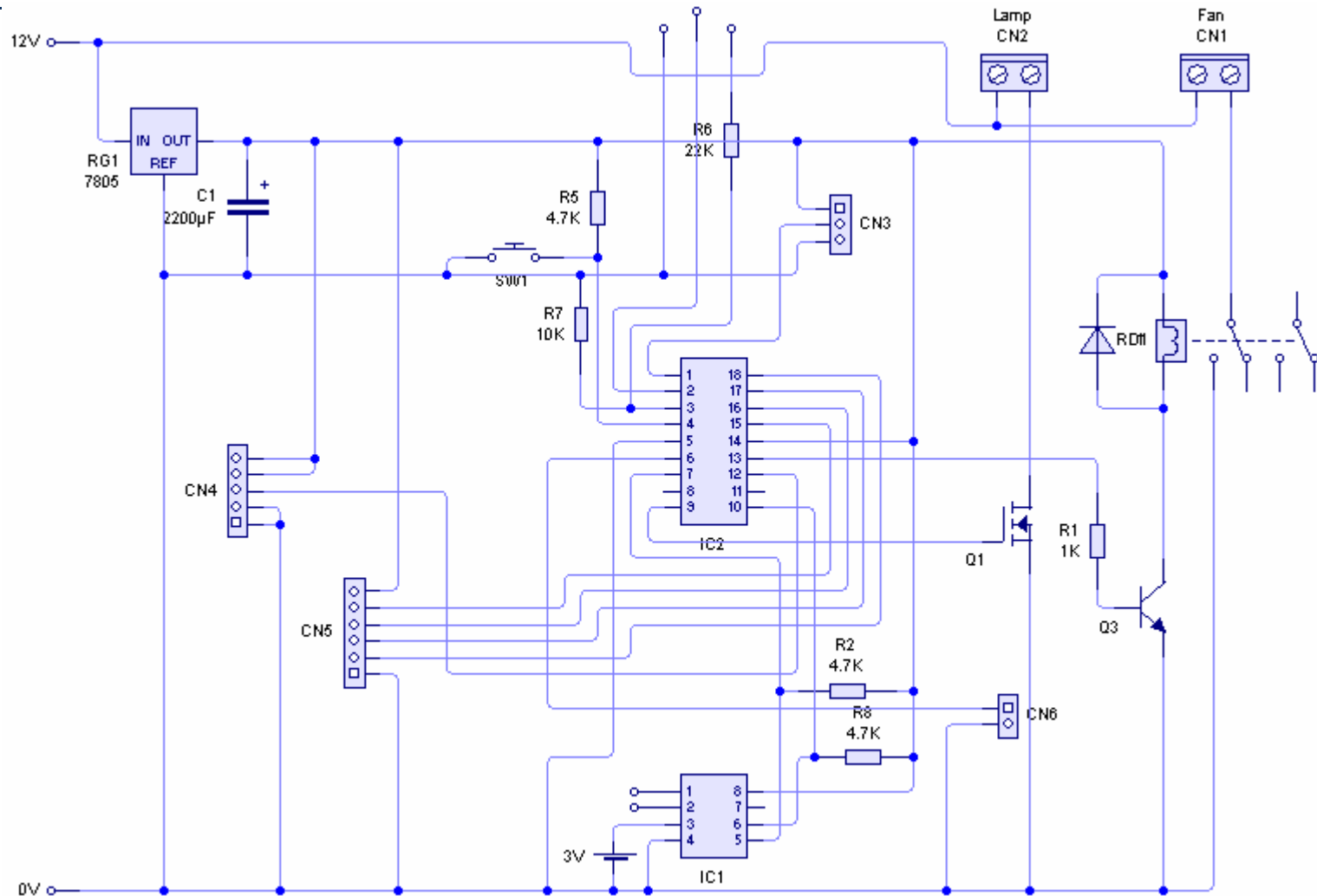
- 1) Completely separate the drive circuitry for the two outputs so that they cannot interfere. In order to do this I would have to use a PICAXE that supports two pwmout pins. The smallest one up from the 18X that does is the 28X and has quite different pin outs. The problem with this is that I would have to completely redesign the artwork and would lose a lot of the benefits of the evolutionary process that the circuit & PCB have gone through.
- 2) Use a switching device in place of the MOSFET that isolates input and output, i.e. a relay. This wouldn't require any change of PICAXE because I could continue using a standard high/low output to drive this. Thus I wouldn't need to completely redesign the artwork, and would keep the risk of tracking errors to a minimum when combined with my colour coding technique.

## Production - Schematic 4.0

I chose the second option that I described on the previous page due to the advantages I described. Shown on the right is my implementation of this new lamp and fan driver circuit.

Since designing schematic 3.0 and artwork 3.0 I had decided what sound output to use, and confirmed that the piezo I chose worked well when I connected it to PCB 2.0 and updated the program.

It was therefore no longer necessary that I provided myself with a 12v or 5v pad for experimentation with different sound output devices, so I removed these from the schematic.



Schematic 4.0

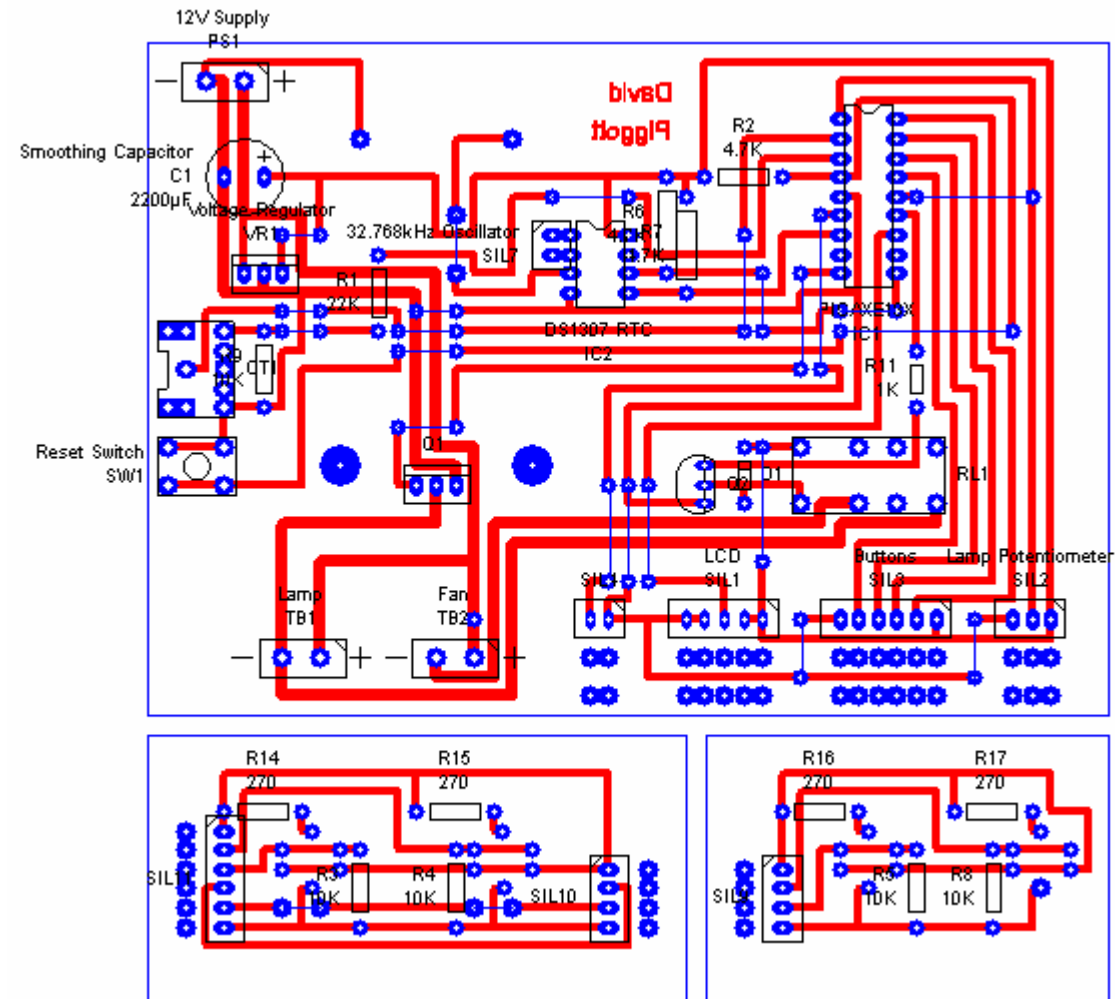
## Production - Artwork 4.0

Artwork 4.0 is pictured opposite. There are three changes from artwork 3.0. The first is that I made the pads for the heatsink mounting holes larger, to allow me to solder the heatsink to the board.

Second is the removal of the 5v and 12v pads I provided for sound output experimentation, for the reason described on the previous page. I also added pads as markers for weaving holes for the piezo wires, which I had forgotten to do on artwork 3.0.

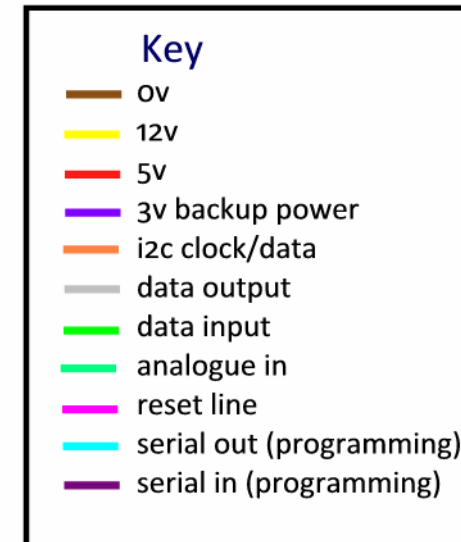
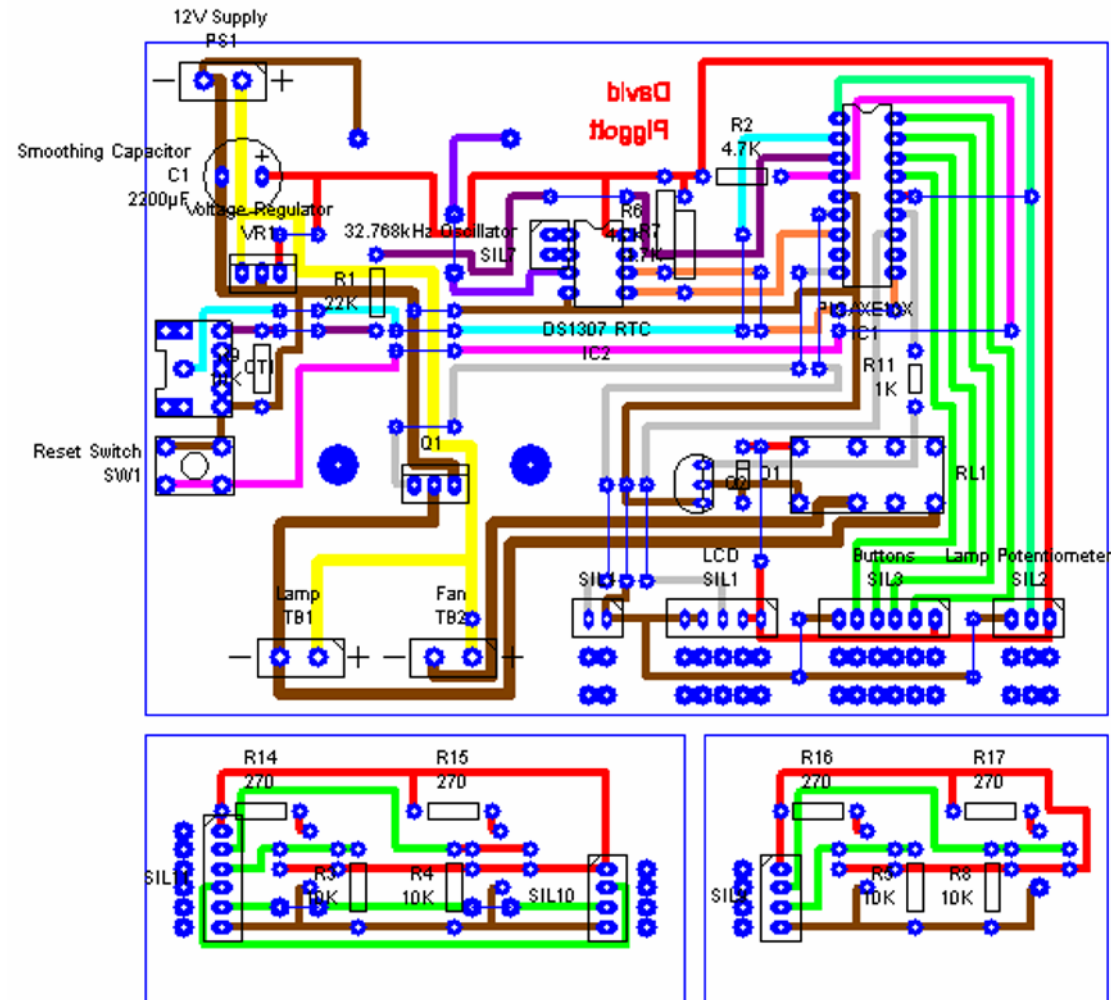
The last and most significant changes is the complete replacement of the fan switching mechanism with a relay instead of the original MOSFET, for reasons already covered. This required a small amount of retracking surrounding the relay area, to make room.

I used the standard relay circuit that I had used during the course.



Artwork 4.0

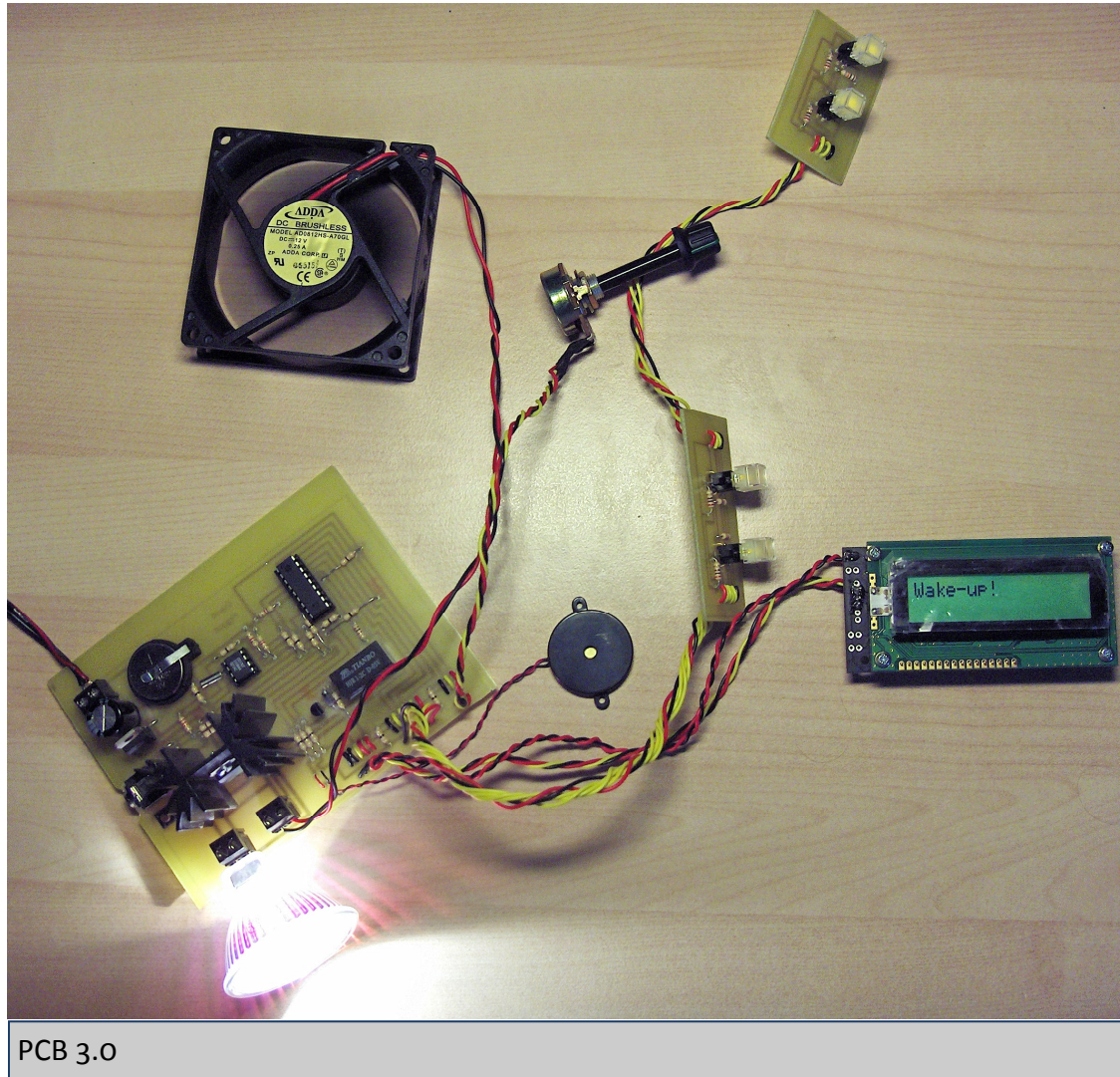
# Production - Artwork 4.0 (Colour Coded)



Once again I have colour coded the artwork, because by doing so I automatically check the artwork for tracking errors. Again I used Microsoft Paint and Microsoft Photodraw.

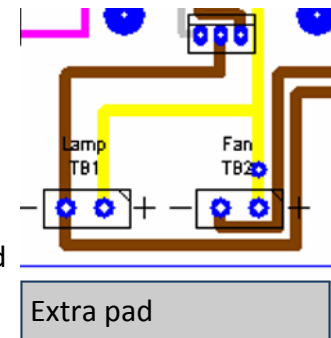


## Production - PCB 3.0 from Artwork 4.0

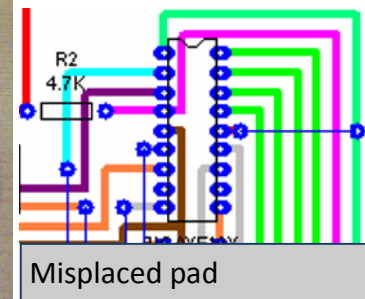


PCB 3.0

Shown on the right is PCB 3.0, produced from artwork 4.0. I noticed two mistakes with the artwork when populating the board, one of which caused a problem.



The first one was a surplus pad placed on the +12v track that goes to the fan terminal block as shown on the right.



The second was that I had misplaced one of the pads for the  $0\Omega$  resistor linking the 5v pin of the PICAXE to a +5v track and instead placed it on the analogue in track.

To fix this I drilled a hole at the correct location on the +5v track and soldered the resistor there. This worked out very well and is hardly noticeable on the board.

As can be seen in the picture, everything works as it should with this PCB!

## Case Development - Design Overview

### Design Overview

I chose to use the first of the ideas I came up with way back before circuit development. A small scan of this is shown on the right as a reminder of the general idea.

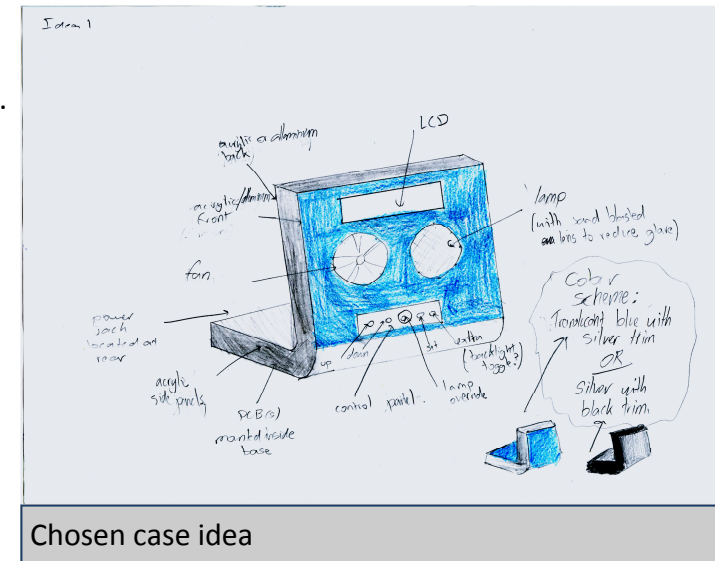
When I came up with this idea, I was not aware that I would need a heatsink on the lamp MOSFET and therefore didn't allow sufficient room for one on the PCB (I had intended for the rear of the base to be no more than approximately 4cm high).

I had originally envisioned an entirely acrylic construction. However there is one flaw in this idea; in order to attach the main panels to the side panels (shown in black in the picture), I would need sufficient surface area for the screws. The 3mm/5mm thickness of acrylic I intended to use certainly wouldn't be enough.

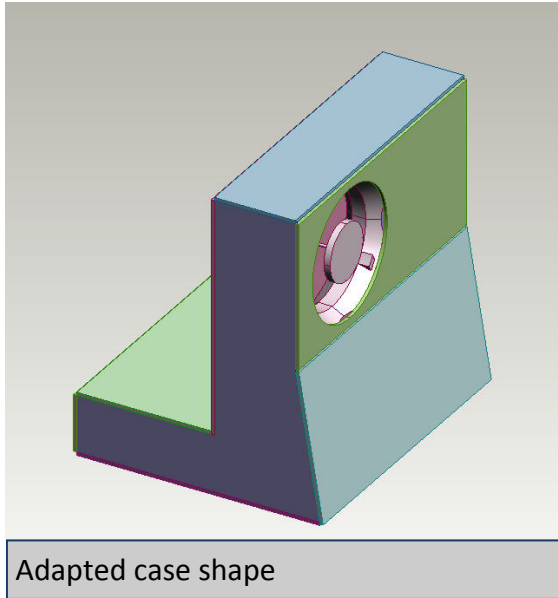
The final problem I would encounter using this idea unadapted concerns the curves along the front between the base panel and front panel. These would require quite a lot of time to implement well and because at this stage in production time was becoming limited, I decided to drop these from the design; since they were not a part of the specification I do not consider this a failing of the case design.

In addition to the above described problems, I decided that the front panel layout of the initial idea was not optimal in terms of user ergonomics; the primary cause for this is the incline of it. I realised that the optimal setup would be one in which the front panel components are perpendicular to the users line of sight; this ensures maximum LCD contrast thus making it easier to read, and also maximum airflow and light intensity thus ensuring maximum effectiveness of the wake-up call.

Depending on whether the alarm clock is placed higher or lower than the user, the front panel should be inclined either forward or backward to ensure it is perpendicular to the users line of sight. Because users will position the alarm clock at different heights the optimal incline will vary. There are two solutions to this problem; a hinged front console, or a compromise incline. I chose the latter.



## Case Development



Adapted case shape

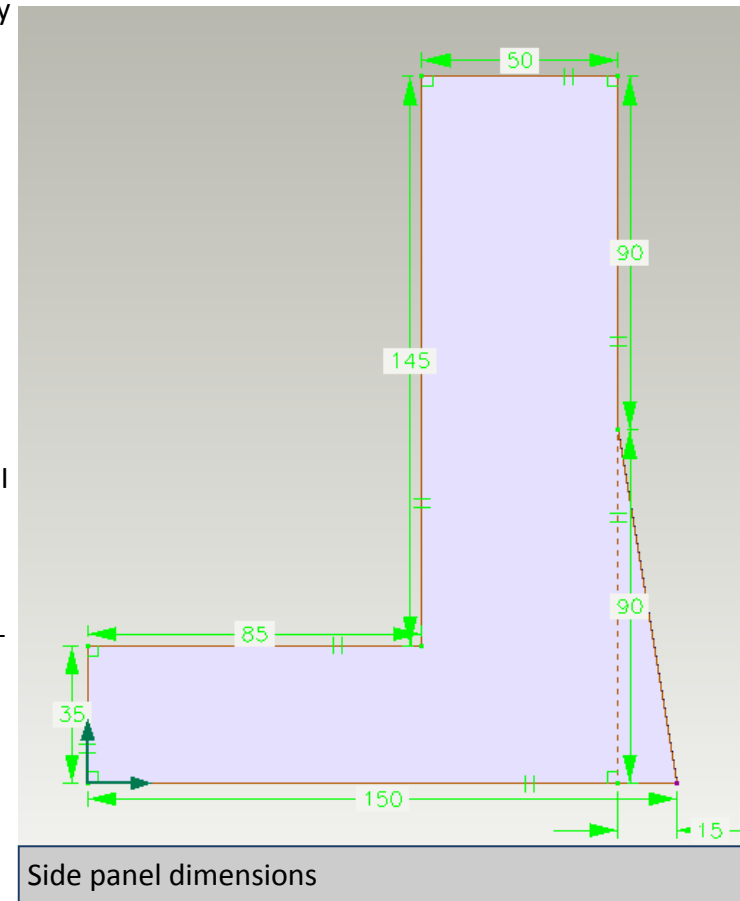
For the reasons described on the previous page I have adapted the case design as illustrated by the model shown below, created using ProDESKTOP. Because at this stage in the project I had already used the 40 hours I allocated for the project, and because the deadline was rapidly approaching, I did not make use of ProDESKTOP to the extent that I would otherwise have done (i.e. create every separate component, dimensioned at every stage, and then assemble them together and produce photorealistic renders).

If I had had time to create all the components on ProDESKTOP, dimensioning the case itself would have been much simpler than it was due to the interference checking function of ProDESKTOP. This function allows you to check for solids that intersect.

By fully dimensioning all the components and using the interference checking feature I would have been able to easily ensure that the design would work and everything would fit in the case.

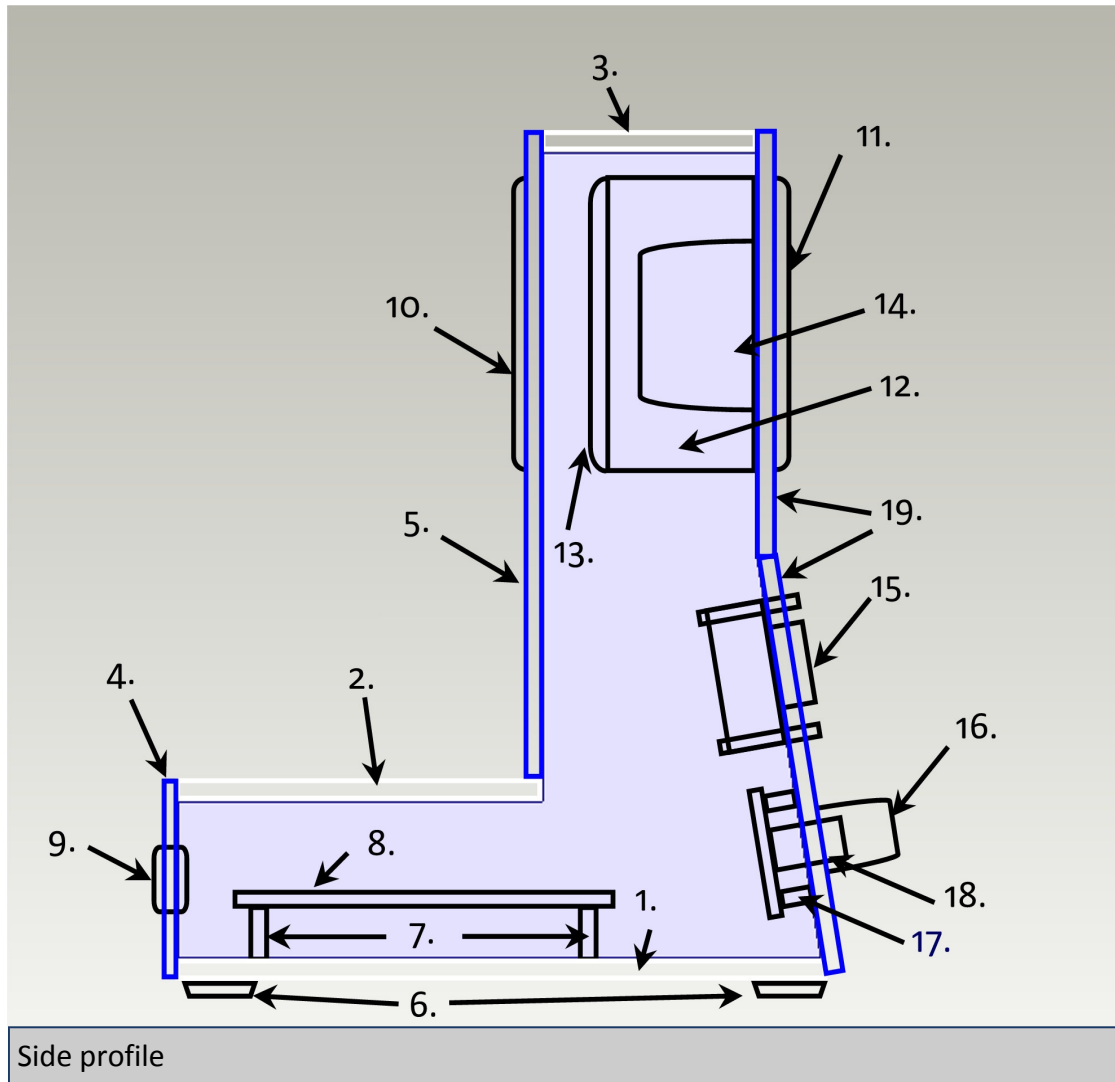
Because I didn't have time to create all the components however I had to use an alternate approach; pencil and paper.

Shown bottom right are the final side profile dimensions I came up with after having created many rough sketches working with the critical dimensions of all the components. These sketches are included for viewing in appendix B. I found that the minimum exterior width, allowing for total deviations of no more than 5mm, was 190mm.



Side panel dimensions

## Case Development - Side Profile

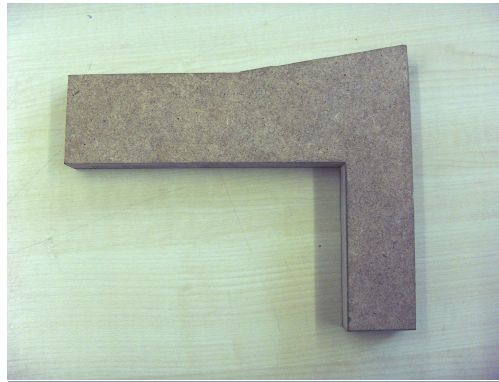


The diagram on the left and the list and key that follows serve two purposes. They detail the order I assembled the case in, and the name of the part referenced:

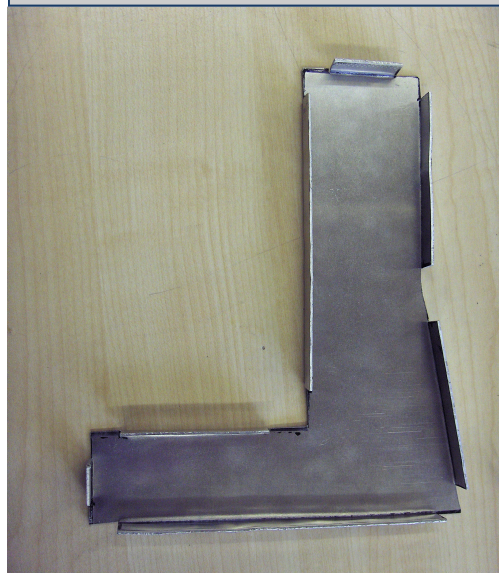
1. Base panel (5mm x 150mm x 190mm clear acrylic)
2. PCB cover panel (5mm x 85mm x 190mm clear acrylic)
3. Top panel (5mm x 50mm x 190mm clear acrylic)
4. Rear panel (3mm x 45mm x 190mm translucent blue acrylic)
5. Mid panel (3mm x 145mm x 190mm translucent blue acrylic)
6. Square rubber feet
7. Nylon PCB (main) standoffs
8. PCB (main)
9. DC power socket
10. Rear fan guard
11. Front fan guard
12. Fan
13. Internal fan guard
14. Lamp
15. LCD
16. Dimmer potentiometer & knob
17. Nylon PCB (switchboard) standoffs
18. PCBs (switchboards)
19. Front console panel (3mm x 200mm x 190mm translucent blue acrylic)



## Case Production



MDF former



Formed side panel

### Aluminium Panel Creation

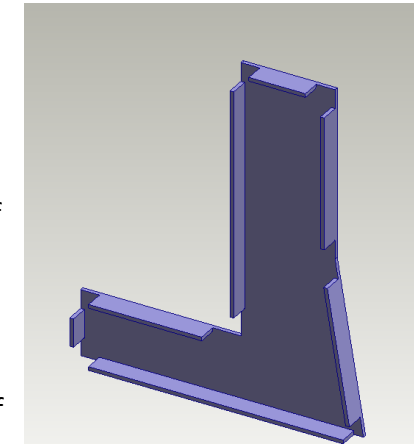
Having recognised need to make the side panels from something other than plastic I decided to use aluminium. The panel component of the ProDESKTOP model I created is shown top right. All tabs are 10mm deep and there is a 10mm gap between either side of the tabs and the edge of the face the tab is attached to.

To construct the panel, a former was made from 18mm thickness MDF, shown top left. I then marked onto 2mm thickness aluminium the net that I had created, using a set square and ruler throughout as a means of quality control. For the large cuts, I used the guillotine to cut the aluminium, and then used the hegner saw for the finer details (between the tabs).

For the smaller tabs, I clamped the tabs in a vice and then bent the panel until it was perpendicular to the tab. For the larger tabs, I clamped the panel between the former shown top left and some scrap blocks of wood. Then using a soft mallet I beat the tabs into the correct shape. The formed panel is shown bottom left.

Considering it was my first attempt I was quite happy with the result. However, I had doubts as to how much I could improve on it in terms of accuracy. In addition to this, the available aluminium sheet had no protective plastic covering on it and as a result was very scratched - which I don't want on the finished case.

Because time was becoming limited with the deadline approaching, I decided that the best compromise would be to make the side panels out of MDF.



ProDESKTOP model

## Case Production

---

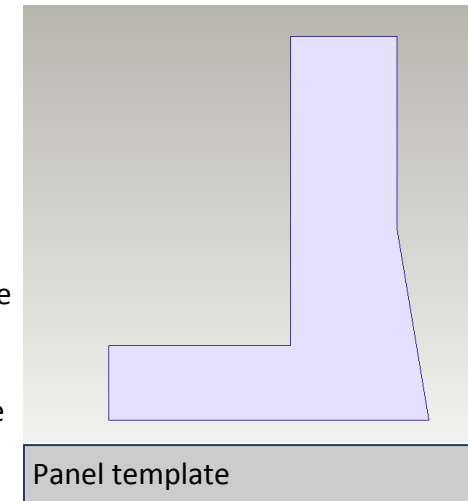
### MDF Panel Creation

Due to the profile I had decided on for the side panels, cutting the MDF was a little more complicated than cutting out a rectangular shape. It was decided that the best way would be to print out a copy of the side profile I had drawn on ProDESKTOP and stick this to the MDF, which would then be cut with the bandsaw. Getting this right took a bit of experimentation due to scaling issues when printing. I printed the profile at 100% zoom level, yet when I measured the lines on paper it was slightly too small. After some experimentation I found that a zoom level of 102% yielded results within 1mm of the sizes of the design. I had a choice of 9, 12 or 18mm thickness MDF.

- 18mm MDF was too thick because it would interfere with the edges of the fan and switchboards.
- 9mm MDF proved too thin after a construction attempt in which it split when the acrylic panels were screwed on.
- 12mm MDF was therefore my final choice.

Shown top right is the template I described, that I printed and stuck to the MDF panels for cutting. Once the panels had been cut, I then had to remove the paper; it had stuck quite well so I had to remove it by sanding the panels until it was removed.

For the surface finish of the MDF panels I decided to use silver vinyl to give a smooth finish. Because of the slight inaccuracy when cutting out the panels (partly due to the slight deviation of the printed profile and partly due to any inaccuracies when the panel was cut), I decided that the best method of cutting the vinyl would not be to cut it using the vinyl cutting machine from the digital profile I created, but instead to use the real MDF panels as templates. In order to do this I simply pressed each of the four large sides of the MDF panels (two for each) against the MDF and cut around it with a scalpel knife.



# Case Production

---

## Console and mid panel creation

I used the Roland CAMM2 milling machine with a 3mm milling tool to create the console and mid panels. This was necessary because of the shapes that needed to be cut;

- The air intake/exhaust holes couldn't be drilled with the pillar drill because of their large diameter; attempting to do so would have just resulted in the acrylic cracking.
- The same is true of the lamp reflector hole.
- The control buttons needed square holes which obviously couldn't be made with the pillar drill.
- The LCD needed a rectangular hole which again couldn't be made with the pillar drill.
- The dimmer potentiometer hole could have been drilled with the pillar drill but since I was using CAMM2 for the panel anyway it made sense to use CAMM2 for this too.

I first created the mid panel, which needed one 80mm diameter hole for the air intake and four 5mm holes surround it, for mounting the fan guard. I used Techsoft 2D design software to draw up the design and plot to the CAMM2 machine with. This was a success.

I then created the front console panel which needed holes for the fan, fan mounting, lamp, LCD, dimmer potentiometer and control buttons. For this I had a 200mm x 220mm acrylic panel cut and then when I drew up the design on 2D design I included a 190mm x 195mm rectangle around the design. This was so that the panel edges would be accurately cut (which isn't so easily achieved when the panels are cut with the bandsaw).

When creating the console panel I had the choice of making two separate panels (one for the lamp and fan, one for the control IO) or a single panel and then bending it with the line bender. I chose the latter because if I had created them as separate panels the gap between them would have been visually unattractive.

See appendix B for the rough sketches from which I worked out the dimensions for the panels.

## Case Production

### Finishing touches to the electronics

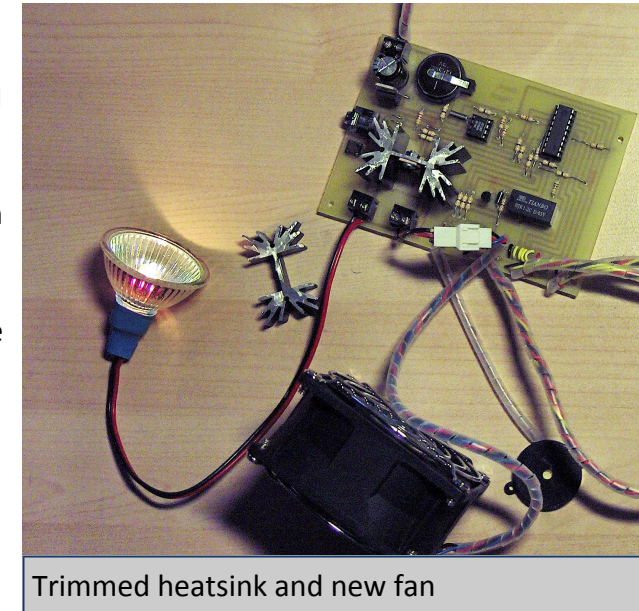
Throughout the circuit & program development I used a fairly cheap and low airflow fan that I got from Rapid through the school. However, right at the beginning of the project when I did my component research, I had found a Delta 80mm high power fan that I was going to use in the project - I hadn't forgotten this, and prior to assembling the case I purchased three 80mm fan guards and the high power Delta fan I had wanted.

The fan is a three wire device (one is for RPM reporting) and I did not wish to destroy the wire termination. I therefore used a three wire male fan connector to connect the fan to the main PCB.

The fan guards are necessary for two reasons; firstly to make the case more aesthetically pleasing, but secondly and more importantly for safety. Due to the extreme power of the Delta fan, there is a risk of serious injury if you were to get your finger in the way of the blades, hence I have restricted access to the blades with the guards. The reason for the internal guard is to prevent the internal wiring from somehow getting in the way of the blades and being shredded by the blades. It also acts a second barrier in the event that the external rear guard is removed.

A few more things had to be done to the PCBs (main and switchboards) before they were ready to be mounted in the case. Firstly I cut them down to the borders I had marked on the artworks, using a Dremel and cutting wheel. I then drilled the holes for the PCB standoffs on which they would be mounted.

The final, and most important thing that had to be done to the PCB was to cut it down to size; the heatsink was originally too large to fit in the case (I was aware of this when designing the case). However rather than make the PCB containing part of the case taller, I decided that it would be better for the case aesthetics to instead trim the heatsink down. I did this with a Dremel and cutting wheel; the whole operation took approx. 60 minutes (including time during which I let the Dremel and heatsink cool down). Shown top right are the finished electronics.



## Final Program - Cutting down the bits

---

Once I had created the breadboard program (documented in the breadboarding & programming section), I continually improved it while waiting on other things during production (e.g. waiting for wood/plastics to be cut). Prior to adding any improvements I first of all had to finish it off; when I wrote the breadboard program, I didn't include any code for the sound output because I didn't have one, and I didn't include any code to control the fan when the wakeup call runs. Rather than detailing every minor program version I made (as this would take literally hundreds of pages if I included the full program each time), I have instead included only the final program.

Between writing the breadboard program and the final program, I discovered the cause of the problem I had encountered with some types of IF statement; I had neglected to enable the enhanced compiler in the setup menu. On doing so I found that I was able to use much more powerful syntax.

The changes from the original program that change functionality are detailed below:

- Addition of code for piezo sounder (used at end of setup menu to provide confirmation of save).
- Beeping during second stage of wake-up call (the 'assertive' mode).
- Reordering of menus, placing all alarm related screens first, followed by the real time and date setup screens. This is because the most commonly altered settings will be those relating to the alarm, so placing these first ensures it is convenient to use.
- Addition of code that allows the fan to be switched on and off when the clock is idle (though control is not independent of the lamp dimmer). The button the code uses is the up button.
- Addition of code that allows the wake-up call to be dismissed during the second stage of the wake-up call (assertive mode). The button that this code monitors is the down button.

However there are more than just changes to the functionality of the program. The compiled size of the original breadboard program is 1899 bytes (out of 2048 bytes available). After I had added the new features detailed above the compiled size was greater than 2048 bytes and thus I couldn't download it to the PICAXE. Therefore drastic changes to the program were necessary. I spent a lot of time planning the changes and have done my best to explain them on the following pages.

At all stages of memory usage optimisation, the question I asked myself when making decisions was 'What will minimise repeated code sections the most?'

## Final Program - Cutting down the bits

---

There is one part of the program that contained excessive amounts of repeated code - the setup menus. Because the repeated code blocks were similar but not identical, I couldn't simply write a subprocedure for them all and be done with it. Instead, more involved programming was necessary.

Despite the limitations of BASIC, I decided to create a system that would mimic the feature of more powerful programming languages to create custom user functions. What I mean by this is that instead of having a standard subprocedure for the repeated code, I created a subprocedure that works with variables wherever the parameters of the repeated code must vary.

Putting the abstract into context, what I am referring to is the increment/decrement code that I wrote for the binary coded decimals. This is used in every single setup screen except for the day, weekend wakeup enable/disable, fade-in time and dismiss time screens. It was therefore quite obvious that by reducing the repetition of code I would be able to significantly reduce the size of the compiled program.

Combined with the fact I had now enabled the enhanced compiler, I was able to rewrite all the BCD inc/dec blocks into the following block. As can be seen, it uses three variables; digit, value and maxvalue. The value of the digit variable when the procedure is called doesn't matter be-

```

if UP = 1 then
    let digit = value & %00001111
    if digit = 9 then
        let digit = value & %01110000
        let value = digit + $10
    else
        let value = value + $01
    endif
elseif DOWN = 1 then
    if value = $00 then
        let value = maxvalue
    else
        let digit = value & %00001111
        if digit = 0 then
            let digit = value & %01110000
            let value = digit - $10
            let value = value + $09
        endif
        let value = value - $01
    endif
    if value > maxvalue then
        let value = $00
    endif
endif

```



## Final Program - Cutting down the bits

cause it is only used by the procedure for temporary storage of the digits.

The value variable is set before the procedure is called. In this way I have created a primitive way of passing parameters to my makeshift function. When the 'function' (subprocedure) returns back to the program block that called, the calling block can then use the value of the appropriately named value variable as necessary.

Despite my pledge to reduce repeated code, this inc/dec function that I created is actually featured in the final program in three different places; once in each of the print blocks (printtime, printalarmtime and printdate). The reason for this is that the necessary bitmasks vary; for the printdate procedure, the bitmask for the tens must be %11110000 to allow the range to reach 99 (as illustrated by the DS1307 register location included again top right).

	BIT7								BIT0	
00H	CH	10 SECONDS		SECONDS						00-59
	0	10 MINUTES		MINUTES						00-59
	0	12 24	10 HR A/P	10 HR		HOURS				01-12 00-23
	0	0	0	0	0	DAY				1-7
	0	0	10 DATE		DATE				01-28/29 01-30 01-31	
	0	0	0	10 MONTH	MONTH				01-12	
	10 YEAR		YEAR						00-99	
07H	OUT	0	0	SQWE	0	0	RS1	RS0		

Register location table

Because my incdec 'functions' required the use of two additional variables, and because I had already used all the available PICAXE byte variables with the breadboard program, I had to completely rethink my use of the available byte variables.

The new memory allocations should be relatively obvious from the symbol definitions at the beginning of the new program. I have defined multiple symbols for the same byte variables; this is because I had to reuse them through the program due to the shortage. This meant I had to be very careful to ensure that at no point in the program does any of the code overwrite variables that are needed before the next poll (in which the important variables are refreshed from the registers of the DS1307).

Because I now had fewer available bytes than before, while having to maintain the same level of functionality, I had to find some spare memory somewhere. The solution to this was through the use of the peek and poke commands of the PICAXE, that enable access to the registers of the PICAXE.

## Final Program - Cutting down the bits

---

In order to reduce the maximum number of simultaneous byte variables required I decided to split the setup menu system into two parts; the alarm setup, and realtime setup.

When the setup menu is loaded, the program reads all settings related to the realtime into the allocated byte variables. These values are immediately stored in the internal registers of the PICAXE at addresses 80 to 86. Then all settings related to the alarm are read into the allocated byte variables (which overlap with the realtime variables, hence the need to store the realtime variables in the internal registers prior to overwriting them). The code for this is as follows:

```

326 setup:
327     readi2c $00,
(setupseconds,setupmins,setuphour,setupday,setupdate,setup
pmonth,setupyear)
328
329     poke 80,setupseconds
330     poke 81,setupmins
331     poke 82,setuphour
332     poke 83,setupday
333     poke 84,setupdate
334     poke 85,setupmonth
335     poke 86,setupyear
336
337     readi2c $08,
(setupalarmmin,setupalarmhour,setupfullweekalarm,setupfa
demins,setupdismissmins)
338
339     serout 6,N2400,(254,1)

```

The setup system then proceeds, advancing through the alarm setup screens when the set button is pressed, and moving back when the exit button is pressed.



## Final Program - Cutting down the bits

---

However when the set button is pressed on the dismissmins screen, bytes must be juggled before the screen can advance to those concerning the real time. The shuntforward block, shown below takes care of this (functioning should be obvious after the initial peek/poke example):

```

270 shuntforward:
271     poke 87,setupalarmmin
272     poke 88,setupalarmhour
273     poke 89,setupfullweekalarm
274     poke 90,setupfademins
275     poke 91,setupdismissmins
276
277     peek 80,setupseconds
278     peek 81,setupmins
279     peek 82,setuphour
280     peek 83,setupday
281     peek 84,setupdate
282     peek 85,setupmonth
283     peek 86,setupyear
284
285     goto sethour

```

Once on the sethour screen, setup continues as normal if moving forward. However if moving back by pressing the exit button it is again necessary to juggle the bytes around, this time with shuntback block:

```

287 shuntback:
288     poke 80,setupseconds
289     poke 81,setupmins
290     poke 82,setuphour
291     poke 83,setupday
292     poke 84,setupdate
293     poke 85,setupmonth
294     poke 86,setupyear
295
296     peek 87,setupalarmmin
297     peek 88,setupalarmhour
298     peek 89,setupfullweekalarm
299     peek 90,setupfademins
300     peek 91,setupdismissmins
301
302     goto setdismissmins

```

When the setup screens are exited by use of the set button, the values must all be saved to the DS1307. The save block takes care of this, shown overleaf.

## Final Program - Cutting down the bits

---

Save block:

```

304 save:
305     serout 6,N2400,(254,1,"Saving...")
306
307     writei2c $00,
(setupseconds,setupmins,setuphour,setupday,setupdate,setup
pmonth,setupyear)
308
309     peek 87,setupalarmmin
310     peek 88,setupalarmhour
311     peek 89,setupfullweekalarm
312     peek 90,setupfademins
313     peek 91,setupdismissmins
314
315     writei2c $08,
(setupalarmmin,setupalarmhour,setupfullweekalarm,setupfa
demins,setupdismissmins)
316
317     sound 0,(32,15)
318     pause 50
319     sound 0,(64,15)
320     pause 50
321     sound 0,(96,15)
322     pause 450
323     serout 6,N2400,(254,1)
324     goto poll

```

The next few pages show the final program. At the beginning of the program, in the polling section, there is a section of code I have commented out:

```

58     'if seconds = 255 then
59     '     serout 6,N2400,(254,1)
60     '     do
61     '         serout 6,N2400,(254,128,"Clock er-
ror!")
62     '         readi2c $00, (seconds)
63     '         loop while seconds = 255
64     '         serout 6,N2400,(254,1)
65     '         goto poll
66     'endif

```

This code is for one of the features I had to drop due to the memory constraints of the PICAXE; it was an error reporting feature that prints "Clock error!" to the LCD in the event that PICAXE cannot communicate with the DS1307.

The other, more significant feature I had to drop, was the option for 12 hour or 24 hour display mode; I got half way to finishing this and then realised the code would be too big to run on the PICAXE.

## Final Program

---

```

1  init:
2      pause 500
3      serout 6,N2400,(254,1)
4      pause 30
5
6      i2cslave %11010000, i2cslow, i2cbyte
7
8      symbol UP = pin1
9      symbol DOWN = pin0
10     symbol SET = pin7
11     symbol BACK = pin6
12
13     symbol fademins = b0
14     symbol fadeintensity = b0
15     symbol dismissmins = b1
16     symbol dismissbeepcounter = b1
17
18     symbol fadepause = w1
19     symbol dismissloopcounter = w1
20     symbol dismisslooplimit = w2
21
22     symbol seconds = b0
23     symbol mins = b1
24     symbol hour = b2
25     symbol day = b3
26     symbol date = b4
27     symbol month = b5
28     symbol year = b6
29     symbol alarmmin = b7
30     symbol alarmhour = b8
31     symbol fullweekalarm = b9
32     symbol fanenable = b10
33
34     symbol setupseconds = b0
35     symbol setupmins = b1
36     symbol setuphour = b2
37     symbol setupday = b3
38     symbol setupdate = b4
39     symbol setupmonth = b5
40     symbol setupyear = b6
41
42     symbol setupalarmmin = b0
43     symbol setupalarmhour = b1
44     symbol setupfullweekalarm = b2
45     symbol setupfademins = b3
46     symbol setupdismissmins = b4
47
48     symbol digit = b11
49     symbol value = b12
50     symbol maxvalue = b13
51
52     poll:
53         readi2c $08,
54         (alarmmin,alarmhour,fullweekalarm,fademins,dismissmins)
55         poke 93,fademins
56         readi2c $00,
57         (seconds,mins,hour,day,date,month,year)
58         'if seconds = 255 then
59         '    serout 6,N2400,(254,1)
60         '    do

```

## Final Program

---

```

61      '      serout 6,N2400,(254,128,"Clock er-
ror!")
62      '      readi2c $00, (seconds)
63      '      loop while seconds = 255
64      '      serout 6,N2400,(254,1)
65      '      goto poll
66      'endif
67
68      if SET = 1 then setup
69      if UP = 1 then
70          if fanenable = 1 then
71              let fanenable = 0
72              low 7
73              serout 6,N2400,(254,128,"Fan dis-
abled!")
74              pause 2000
75          else
76              let fanenable = 1
77              high 7
78              serout 6,N2400,(254,128,"Fan en-
abled!")
79              pause 2000
80          endif
81      endif
82
83      if day = 7 or day = 1 then
84          if fullweekalarm = 1 then
85              if mins = alarmmin and hour = alarm-
hour then goto wakeup
86              endif
87          else
88              if mins = alarmmin and hour = alarmhour
then goto wakeup
89              endif
90
91      clock:
92          serout 6,N2400,(254,192)
93          gosub printdate
94          serout 6,N2400,(254,128)
95          gosub printtime
96
97          goto poll
98
99      lamp:
100         readadc10 2,w4
101         pwmout 3,249,w4
102         return
103
104     printtime:
105         gosub lamp
106
107         let digit = hour & %00110000 / 16
108         serout 6,N2400,(#digit)
109         let digit = hour & %00001111
110         serout 6,N2400,(#digit,":")
111
112         let digit = mins & %01110000 / 16
113         serout 6,N2400,(#digit)
114         let digit = mins & %00001111
115         serout 6,N2400,(#digit,":")
116
117         let digit = seconds & %01110000 / 16

```

## Final Program

---

```

118  serout 6,N2400,(#digit)
119  let digit = seconds & %00001111
120  serout 6,N2400,(#digit," ")
121
122  if UP = 1 then
123      let digit = value & %00001111
124      if digit = 9 then
125          let digit = value & %01110000
126          let value = digit + $10
127      else
128          let value = value + $01
129      endif
130  elseif DOWN = 1 then
131      if value = $00 then
132          let value = maxvalue
133      else
134          let digit = value & %00001111
135          if digit = 0 then
136              let digit = value & %01110000
137              let value = digit - $10
138              let value = value + $09
139          endif
140          let value = value - $01
141      endif
142  endif
143  if value > maxvalue then
144      let value = $00
145  endif
146
147  return
148
149  printdate:
150      gosub lamp
151
152      let digit = date & %00110000 / 16
153      serout 6,N2400,(#digit)
154      let digit = date & %00001111
155      serout 6,N2400,(#digit,"/")
156
157      let digit = month & %00010000 / 16
158      serout 6,N2400,(#digit)
159      let digit = month & %00001111
160      serout 6,N2400,(#digit,"/")
161
162      let digit = year & %11110000 / 16
163      serout 6,N2400,("20",#digit)
164      let digit = year & %00001111
165      serout 6,N2400,(#digit," ")
166
167  if UP = 1 then
168      let digit = value & %00001111
169      if digit = 9 then
170          let digit = value & %11110000
171          let value = digit + $10
172      else
173          let value = value + $01
174      endif
175  elseif DOWN = 1 then
176      if value = $01 then
177          let value = maxvalue
178      else
179          let digit = value & %00001111

```

## Final Program

---

```

180         if digit = 0 then
181             let digit = value & %11110000
182             let value = digit - $10
183             let value = value + $09
184         endif
185         let value = value - $01
186     endif
187 endif
188 if value > maxvalue then
189     let value = $01
190 endif
191 return
192
193 printalarmtime:
194     gosub lamp
195
196     let digit = setupalarmhour & %00110000 / 16
197     serout 6,N2400,(#digit)
198     let digit = setupalarmhour & %00001111
199     serout 6,N2400,(#digit,":")
200
201     let digit = setupalarmmin & %01110000 / 16
202     serout 6,N2400,(#digit)
203     let digit = setupalarmmin & %00001111
204     serout 6,N2400,(#digit," ")
205
206     if UP = 1 then
207         let digit = value & %00001111
208         if digit = 9 then
209             let digit = value & %01110000
210
211             let value = digit + $10
212         else
213             let value = value + $01
214         endif
215     elseif DOWN = 1 then
216         if value = $00 then
217             let value = maxvalue
218         else
219             let digit = value & %00001111
220             if digit = 0 then
221                 let digit = value & %01110000
222                 let value = digit - $10
223                 let value = value + $09
224             endif
225             let value = value - $01
226         endif
227     endif
228     if value > maxvalue then
229         let value = $00
230     endif
231
232     return
233
234 wakeup:
235     peek 93,fademics
236     peek 94,dismissmins
237
238     serout 6,N2400,(254,128,"Wake Up!")
239     high 7
240
241     let fadepause = fademics * 300

```

## Final Program

---

```

242   let fadeintensity = 0
243   for fadeintensity = 0 to 200
244       pwmout 3,49,fadeintensity
245       pause fadepause
246   next fadeintensity
247
248   let dismisslooplimit = dismissmins * 40
249   let dismissloopcounter = 0
250   for dismissloopcounter = 0 to dismisslooplimit
251       let dismissbeepcounter = 0
252       for dismissbeepcounter = 0 to 3
253           sound 0,(96,15)
254           pause 100
255           if DOWN = 1 then
256               serout 6,N2400,
(254,128,"Dismissed!")
257               low 7
258               pause 2000
259               goto poll
260           endif
261       next dismissbeepcounter
262       pause 500
263   next dismissloopcounter
264   pwmout 3,0,0
265
266   serout 6,N2400,(254,1)
267   low 7
268   goto poll
269
270 shuntforward:
271   poke 87,setupalarmmin
272   poke 88,setupalarmhour
273   poke 89,setupfullweekalarm
274   poke 90,setupfademins
275   poke 91,setupdismissmins
276
277   peek 80,setupseconds
278   peek 81,setupmins
279   peek 82,setuphour
280   peek 83,setupday
281   peek 84,setupdate
282   peek 85,setupmonth
283   peek 86,setupyear
284
285   goto sethour
286
287 shuntback:
288   poke 80,setupseconds
289   poke 81,setupmins
290   poke 82,setuphour
291   poke 83,setupday
292   poke 84,setupdate
293   poke 85,setupmonth
294   poke 86,setupyear
295
296   peek 87,setupalarmmin
297   peek 88,setupalarmhour
298   peek 89,setupfullweekalarm
299   peek 90,setupfademins
300   peek 91,setupdismissmins
301
302   goto setdismissmins

```



## Final Program

---

```

303
304 save:
305     serout 6,N2400,(254,1,"Saving...")
306
307     writei2c $00,
(setupseconds,setupmins,setuphour,setupday,setupdate,setup
pmonth,setupyear)
308
309     peek 87,setupalarmmin
310     peek 88,setupalarmhour
311     peek 89,setupfullweekalarm
312     peek 90,setupfademins
313     peek 91,setupdismissmins
314
315     writei2c $08,
(setupalarmmin,setupalarmhour,setupfullweekalarm,setupfad
emins,setupdismissmins)
316
317     sound 0,(32,15)
318     pause 50
319     sound 0,(64,15)
320     pause 50
321     sound 0,(96,15)
322     pause 450
323     serout 6,N2400,(254,1)
324     goto poll
325
326 setup:
327     readi2c $00,
(setupseconds,setupmins,setuphour,setupday,setupdate,setup
pmonth,setupyear)
328
329     poke 80,setupseconds
330     poke 81,setupmins
331     poke 82,setuphour
332     poke 83,setupday
333     poke 84,setupdate
334     poke 85,setupmonth
335     poke 86,setupyear
336
337     readi2c $08,
(setupalarmmin,setupalarmhour,setupfullweekalarm,setupfa
demins,setupdismissmins)
338
339     serout 6,N2400,(254,1)
340
341 setalarmhour:
342     serout 6,N2400,(254,128,"Alarm Hour:
",254,192)
343
344     let value = setupalarmhour
345     let maxvalue = $23
346
347     gosub printalarmtime
348
349     let setupalarmhour = value
350
351     if SET = 1 then setalarmmin
352     if BACK = 1 then poll
353
354     pause 100
355

```

## Final Program

---

```

356     goto setalarmhour
357
358 setalarmmin:
359     serout 6,N2400,(254,128,"Alarm Minutes:
",254,192)
360
361     let value = setupalarmmin
362     let maxvalue = $59
363
364     gosub printalarmtime
365
366     let setupalarmmin = value
367
368     if SET = 1 then setfullweekalarm
369     if BACK = 1 then setalarmhour
370
371     pause 100
372
373     goto setalarmmin
374
375 setfullweekalarm:
376     serout 6,N2400,(254,128,"Weekend Wakeup:")
377     if setupfullweekalarm = 0 then
378         serout 6,N2400,(254,192,"Disabled  ")
379     else
380         serout 6,N2400,(254,192,"Enabled  ")
381     endif
382     if UP = 1 or DOWN = 1 then
383         if setupfullweekalarm = 0 then
384             let setupfullweekalarm = 1
385         else
386             let setupfullweekalarm = 0
387         endif
388     endif
389
390     if SET = 1 then setfademins
391     if BACK = 1 then setalarmmin
392
393     pause 100
394
395     goto setfullweekalarm
396
397 setfademins:
398     if setupfademins > 60 then
399         let setupfademins = 0
400     endif
401     serout 6,N2400,(254,128,"Fade-in time:
",254,192,#setupfademins," minute(s) ")
402
403     if UP = 1 then
404         inc setupfademins
405     elseif DOWN = 1 then
406         if setupfademins = 0 then
407             let setupfademins = 60
408         else
409             dec setupfademins
410         endif
411     endif
412     if SET = 1 then setdismissmins
413     if BACK = 1 then setfullweekalarm
414
415     pause 100

```

## Final Program

---

```

416                                     446
417     goto setfademins                 447     gosub printtime
418                                     448
419 setdismissmins:                     449     let setuphour = value
420     if setupdismissmins > 60 then    450
421         let setupdismissmins = 0    451     if SET = 1 then setmins
422     endif                             452     if BACK = 1 then shuntback
423     serout 6,N2400,(254,128,"Dismiss time:  453
",254,192,#setupdismissmins," minute(s) ") 454     pause 100
424                                     455
425     if UP = 1 then                   456     goto sethour
426         inc setupdismissmins        457
427     elseif DOWN = 1 then             458 setmins:
428         if setupdismissmins = 0 then 459     serout 6,N2400,(254,128,"Minutes:",254,192)
429             let setupdismissmins = 60 460
430         else                          461     let value = setupmins
431             dec setupdismissmins     462     let maxvalue = $59
432         endif                         463
433     endif                             464     gosub printtime
434     if SET = 1 then shuntforward      465
435     if BACK = 1 then setfademins     466     let setupmins = value
436                                     467
437     pause 100                         468     if SET = 1 then setseconds
438                                     469     if BACK = 1 then sethour
439     goto setdismissmins              470
440                                     471     pause 100
441 sethour:                             472
442     serout 6,N2400,(254,128,"Hour:         ",254,192) 473     goto setmins
443                                     474
444     let value = setuphour             475 setseconds:
445     let maxvalue = $23                476     serout 6,N2400,(254,128,"Seconds:",254,192)

```

## Final Program

---

```

477
478     let value = setupseconds
479     let maxvalue = $59
480
481     gosub printtime
482
483     let setupseconds = value
484
485     if SET = 1 then setdayclear
486     if BACK = 1 then setmins
487
488     pause 100
489
490     goto setseconds
491
492 setdayclear:
493     serout 6,N2400,(254,1)
494
495 setdaydisp:
496     serout 6,N2400,(254,128,"Day:")
497     if day = $01 then sunday
498     if day = $02 then monday
499     if day = $03 then tuesday
500     if day = $04 then wednesday
501     if day = $05 then thursday
502     if day = $06 then friday
503     if day = $07 then saturday
504
505 sunday:
506     serout 6,N2400,(254,192,"Sunday  ")
507     goto setdaypoll
508 monday:
509     serout 6,N2400,(254,192,"Monday  ")
510     goto setdaypoll
511 tuesday:
512     serout 6,N2400,(254,192,"Tuesday  ")
513     goto setdaypoll
514 wednesday:
515     serout 6,N2400,(254,192,"Wednesday")
516     goto setdaypoll
517 thursday:
518     serout 6,N2400,(254,192,"Thursday ")
519     goto setdaypoll
520 friday:
521     serout 6,N2400,(254,192,"Friday  ")
522     goto setdaypoll
523 saturday:
524     serout 6,N2400,(254,192,"Saturday ")
525     goto setdaypoll
526
527 setdaypoll:
528     pause 100
529     if setupday > 7 then resetday
530     if setupday < 1 then fullday
531     if UP = 1 then incday
532     if DOWN = 1 then decday
533     if SET = 1 then setdate
534     if BACK = 1 then setseconds
535     goto setdaydisp
536
537 incday:
538     let setupday = setupday + 1

```

# Final Program

---

```

539     goto setdaydisp
540 decday:
541     let setupday = setupday - 1
542     goto setdaydisp
543 resetday:
544     let setupday = 1
545     goto setdaydisp
546 fullday:
547     let setupday = 7
548     goto setdaydisp
549
550 setdate:
551     serout 6,N2400,(254,128,"Date:   ",254,192)
552
553     let value = setupdate
554     let maxvalue = $31
555
556     gosub printdate
557
558     let setupdate = value
559
560     if SET = 1 then setmonth
561     if BACK = 1 then setdayclear
562
563     pause 100
564
565     goto setdate
566
567 setmonth:
568     serout 6,N2400,(254,128,"Month:  ",254,192)
569
570     let value = setupmonth
571     let maxvalue = $12
572
573     gosub printdate
574
575     let setupmonth = value
576
577     if SET = 1 then setyear
578     if BACK = 1 then setdate
579
580     pause 100
581
582     goto setmonth
583
584 setyear:
585     serout 6,N2400,(254,128,"Year:   ",254,192)
586
587     let digit = date & %00110000 / 16
588     serout 6,N2400,(#digit)
589     let digit = date & %00001111
590     serout 6,N2400,(#digit,"/")
591
592     let digit = month & %00010000 / 16
593     serout 6,N2400,(#digit)
594     let digit = month & %00001111
595     serout 6,N2400,(#digit,"/")
596
597     let digit = year & %11110000 / 16
598     serout 6,N2400,("20",#digit)
599     let digit = year & %00001111
600     serout 6,N2400,(#digit," ")

```

# Final Program

---

```

601                                     632      goto setyear
602      if UP = 1 then
603          let digit = setupyear & %00001111
604          if digit = 9 then
605              let digit = setupyear & %11110000
606              let setupyear = digit + $10
607          else
608              let setupyear = setupyear + $01
609          endif
610      elseif DOWN = 1 then
611          if setupyear = $00 then
612              let setupyear = $99
613          else
614              let digit = setupyear & %00001111
615              if digit = 0 then
616                  let digit = setupyear & %11110000
617                  let setupyear = digit - $10
618                  let setupyear = setupyear + $09
619              endif
620              let setupyear = setupyear - $01
621          endif
622      endif
623      if setupyear > $99 then
624          let setupyear = $00
625      endif
626
627      if SET = 1 then save
628      if BACK = 1 then setmonth
629
630      pause 100
631

```



## Finished Product - Pictures

---



Finished product picture 1



Finished product picture 2



## Finished Product - Pictures

---



Finished product picture 3



Finished product picture 4

## Time Management - Diary

---

I have detailed all the problems I encountered and my solutions throughout documentation - the only information not included by it is that regarding time. The Gaant chart on the following page shows my usage of time.

One thing the Gaant chart doesn't show is the rate at which I developed programs, because I have shown 'breadboarding and programming to specification' as one item. Infact, once I had recognised that the DS1307s registers contained BCDs I wrote the whole program for the bread-board version on 10/02/07.

The same is true of the program optimisation and new features I added on the last day of the project; though I had done a few days planning before writing the code, this was pretty much an entire re-write of the program (specifically the setup code - which is the majority of the program).

As can be seen I was very quick at producing PCBs once I was aware of problems with previous designs. For example I produced PCB 1 on the 26/02/07. Once aware of problems I managed to update the artwork, expose, develop, etch, drill, populate, solder and test PCB 2 all on the following day - very high workrate!

The table, bottom right, shows a breakdown of the estimated time spent on the project.

Task	Completion time estimate
Paperwork	30 hours
Case ideas	2 hours
Breadboarding	10 hours
Programming to specification	20 hours
Evaluation of electronics and program	2 hours
PCB artwork design	6 hours
PCB production	15 hours
Chosen case idea development	3 hours
Aluminium side panel attempt	2 hours
Case construction and assembly	10 hours
Creation of front and rear panels with CAMM2	2 hours
Preparation of PCB for mounting in case	2 hours
Program rewriting for memory efficiency to enable...	10 hours
...Addition of new features to program	2 hours
Mounting of electronics within case	4 hours
<b>Total</b>	<b>120 hours</b>

# Time Management - Diary

ID	Task Name	Start	Finish	Duration	Jan 2007				Feb 2007				Mar 2007			
					7/1	14/1	21/1	28/1	4/2	11/2	18/2	25/2	4/3	11/3	18/3	
1	Ongoing paperwork	08/01/2007	25/03/2007	77d												
2	Breadboarding & programming to specification	11/01/2007	10/02/2007	31d												
3	Case ideas	15/01/2007	21/01/2007	7d												
4	Evaluation of electronics & program	10/02/2007	25/03/2007	44d												
5	Artwork 1.0 Design	11/02/2007	11/02/2007	1d												
6	Artwork 2.0 Design	25/02/2007	25/02/2007	1d												
7	PCB 1.0 Production from Artwork 2.0	26/02/2007	26/02/2007	1d												
8	Artwork 3.0 Design	27/02/2007	27/02/2007	1d												
9	PCB 2.0 Production from Artwork 3.0	27/02/2007	27/02/2007	1d												
10	Case idea development	28/02/2007	16/03/2007	17d												
11	Aluminium side panel attempts	10/03/2007	11/03/2007	2d												
12	Case construction & assembly	12/03/2007	23/03/2007	12d												
13	Artwork 4.0 Design	17/03/2007	17/03/2007	1d												
14	PCB 3.0 Production from Artwork 4.0	17/03/2007	17/03/2007	1d												
15	Creation of front and rear panels with CAMM2	20/03/2007	23/03/2007	4d												
16	Preparation of PCB for mounting in case	22/03/2007	22/03/2007	1d												
17	Addition of new features to program	23/03/2007	23/03/2007	1d												
18	Mounting of electronics within case	23/03/2007	23/03/2007	1d												

# Evaluation & Testing

---

## Testing

As shown on the Gaant chart showing my actual time usage during the project, there is a 44 day long "Evaluation of electronics & program" starting 10/02/07 and finishing one day before the project deadline.

During this evaluation stage I took the electronics home several times and used the alarm clock on five different mornings instead of my usual alarm clock. It worked brilliantly everytime, and as it was intended, the slow increase in light intensity prevented me from going back to sleep.

I tested the first two PCBs once each and the final PCB three times, with a range of different fade-in and auto-dismiss times.

Unfortunately I wasn't able to test the breeze simulation aspect as thoroughly because I only solved all the problems with the fan control on the final PCB. Even if it weren't for that, my testing would have been irrelevant because I upgraded the low power fan I had used during development with the high power Delta fan.

As I detailed during case development, the issue I had been wary of was the positioning and incline of the front panel components. Having completed and tested the case, with the electronics in, I am glad to say that the design I settled on works very well and I don't think it could be improved upon.

## Faults

I only found one fault with the final PCB. It is an intermittent fault that occurs sometimes during the wake-up call or when the alarm clock is used as a bedside lamp. The problem that occurs is that PICAXE 'crashes' and the alarm clock ceases to function until reset.

I have thoroughly analysed the program to check for bugs and have concluded it bug free; if it had been a software problem I would expect it to occur every time as opposed to being intermittent anyway. The crashes only ever occur at relatively low lamp intensity (i.e. below 1/3 maximum brightness); beyond this it never crashes. I believe the problem to be either:

- Electrical noise due to the PWM used.
- The power supply not being able to cope with the inrush current to the lamp when it is switched on.

# Evaluation & Testing

---

## Improvements

As a result of the extensive testing I did on the electronics, I recognised five areas in which the program could be improved:

1. Right up until I finalised the program, I the setup menu order had the time & date setup first followed by the alarm setup. In testing the alarm clock I recognised that once the user has initially set the time & date they will rarely need to change it, while they will change the alarm settings more often. Therefore it made sense from a user convenience point of view to make the alarm setup first followed by the time & date setup.
2. I added the save confirmation tones when the setup menus are exited and values saved, and a one second wait to prevent the user from immediately entering setup again.
3. I had originally written the sound part of the wake-up call code block so that the piezo beeped continually with a constant interval; this didn't sound much like an alarm clock so I rewrote the sound code so that the piezo beeps four times in quick succession, pauses, beeps four times in quick succession, pauses, and continues in this way.
4. I added code that would enable the fan to be switched on in addition to the lamp, so the alarm clock not only doubles up as a bedside lamp but a fan aswell!
5. Despite having decided at the start of the project not to implement a dismiss button of any sort, I decided that during the second part of the wake-up call, a dismiss button would be overall a good feature. This is a good compromise between forcing the wake-up call on the user (i.e. they can't escape the slow increase in fan and lamp intensity to full) and allowing them to dismiss it.

## General

Overall I am very pleased with the finished product and am glad I chose to create a 'Heavy Sleeper's Alarm Clock'. I have no doubts that my project is an effective, easy to use, and well featured alarm clock. With a few improvements and modifications I believe it could be turned into a retailable product.



# Evaluation & Testing

---

## Evaluation against specification

Below follows my original specification and my evaluation of each point.

### Wake-Up Call - Specification Points

1. A high power 12V lamp will gradually increase in intensity during the wake-up call over an adjustable period of 0 to 60 minutes that I refer to as the fade-in time. Setting the time period to 0 minutes will cause the lamp to go straight to full intensity.
2. An 80mm diameter 12V DC fan will gradually increase in intensity during the wake-up call over the same adjustable period as the lamp.
3. Once at full intensity, the alarm clock will begin the failsafe wake-up call; a loud piezo sounder. This will continue for a user configured time period that I refer to as the dismiss time.
4. Only one alarm time will be supported to simplify use and development.

### Wake-Up Call - Evaluation

1. The lamp used is a 12V lamp and is very bright! The lamp does increase in intensity over the period specified, will switch on immediately if the period is set to 0, and the period is adjustable from 0 to 60 minutes. However the program does occasionally crash due to either electrical noise or the power supply being underpowered. The intensity of the lamp is overridden by the program when the wake-up call runs so it is not a problem if the user has left the lamp on overnight - there will still be a change in light which should wake them.
2. An 80mm fan increases in intensity at the same time as the lamp and is definitely powerful enough to simulate a strong breeze at full intensity. The fan does not switch on at the same time as the lamp however because there is a threshold voltage below which it won't operate. I do not consider this a problem because below this voltage the breeze wouldn't be noticeable anyway. The intensity of the fan is overridden by the program when the wake-up call runs so leaving it on overnight shouldn't cause a problem as there will still be a change in intensity which should wake them.
3. The wake-up call continues for a user configured period of time, adjustable from 0 to 60 minutes, during which a piezo sounder beeps. The piezo I used was the loudest one I could find without a built in drive circuit and using one with a built in drive circuit wasn't an option because I needed to be able to control the tone.
4. Only one time is supported. However, I have added what I consider to be a very useful feature on alarm clocks; the option to disable the wake-up call from running at weekends and I am very pleased with my implementation of this.

# Evaluation & Testing

---

## Technical - Specification Points

1. The alarm clock will have an LCD display to show the current time and for feedback during alarm and time setting.
2. It will use the Serial LCD Module AXE033 to provide the display functionality.
3. It will use the AXE034 clock upgrade to provide the clock functionality.
4. A PICAXE will be used as the core of the system (setting time and alarm time, running the wake up call etc.).
5. An X version of the appropriate size PICAXE will be used because I2C support will be needed to communicate with the clock chip and I anticipate that my program will be quite long (the X parts have more program memory and the necessary I2C command support).
6. The alarm clock will be powered by a 12V DC external power supply (batteries aren't sufficient because of the lamp).
7. The correct time will be maintained when external power is disconnected by the backup battery in the AXE034 clock module.

## Technical - Evaluation

1. The alarm clock uses an LCD display to show the current time and this has also allowed me to create a very effective and easy to use setup system. The LCD has good visibility and the backlight ensures it can be read in the dark.
2. I used the Serial LCD Module AXE033 as a serial driver for another LCD; a 16 x 2 backlit module that I upgraded it with.
3. I used the parts of the AXE034 clock upgrade to provide the clock functionality (i.e. the DS1307 and 3v backup cell). However, I didn't use them with the AXE033 module for the reasons well described in the development stage (need for I2C and serial communications).
4. I used a PICAXE as the core of the system and it was very useful when developing the program thanks to ease with which new programs can be downloaded to the PICAXE.
5. I used an X version of the appropriate size PICAXE (18X). Not only did I need I2C support, but I also needed pwmout support, which the 18X provides. My anticipation about the long program was correct and in fact I had to cut some of the features I would otherwise have added to the program in order to fit it on the 18X - even after thorough optimisation of repeated sections of code (features I would have added include 12hr/24hr option, and an error message on the LCD in the event that the PICAXE cannot communicate with the DS1307).
6. The alarm clock is powered by a 12V DC external power supply which functions correctly most of the time. However I have a suspicion that the power supply may be the cause of an intermittent crash when the wake-up call runs, so I may have chosen an underpowered PSU.
7. The correct time is maintained when external power is disconnected by means of a 3v backup cell connected to the DS1307. Not only is the correct time maintained but I have also made use of the spare RAM available on the DS1307 to store all alarm related settings so



# Evaluation & Testing

---

that when power is restored the users alarm settings are still present (alarm time, weekend wakeup option, fade-in time and dismiss-time).

## Control - Specification Points

1. The alarm clock will have no more than five control inputs, to ensure ease of use.
2. All input and output devices will be located on the front face of the case, so that the outputs are effectively directed at the user, and they can easily reach the control panel.
3. To ensure usability in the dark, the control inputs will be illuminated.
4. Rather than having a snooze or dismiss button, the clock should automatically dismiss itself after both gentle and fail-safe wake-up calls have completed. This is to prevent determined users from going back to sleep by dismissing it (unless of course they unplug the power supply).
5. There will be no on/off switch because the functionality of an on/off switch is simply not necessary with clocks - they are always on.

## Control - Evaluation

1. The alarm clock has exactly five control inputs including the dimmer potentiometer. However I would discount the dimmer potentiometer because it's purpose is so obvious and instead say the alarm clock has only four control inputs. Either way, the setup menu system is very easy to use and for an alarm clock with so many features and options, possibly the easiest to use realistic setup implementation. I am very pleased with it.
2. All input and output devices are located on the front of the case, and I am satisfied that the incline of the various components is optimal in terms of ergonomics.
3. The control buttons are pleasantly illuminated and easy to see in the dark. However there are no illuminated labels, and the dimmer potentiometer is not illuminated. The alarm clock can still easily be used in the dark though because it is easy to remember the function of the buttons, and that the dimmer potentiometer is located centrally between the down and exit buttons.
4. The clock automatically dismisses itself after both the gentle and fail-safe wake-up calls have completed. However I did go against the original specification and implement code so that the up button can also be used to dismiss the alarm clock during the second part of the wake-up call. I believe this to be better than not having a dismiss button at all as this could put some consumers off buying the product, and without one people would just unplug it anyway.

## Evaluation & Testing

---

5. There is no on/off switch and this was definitely the correct decision - there is simply no need for one.

### Case - Specification Points

1. The case should be as small as is reasonably possible given the output components I will be using, so that it isn't imposing on the room.
2. All components will be contained within one case—that is, the dawn simulation lamp and breeze fan will be housed within the main case, to ensure convenience when setting up and positioning the alarm clock.
3. Connection to the power supply will be by a DC power jack located at the rear of the case.
4. The circuit board will be firmly attached inside the case using stand-offs.
5. The alarm clock will also double up as a nightlight, so will require a lamp override control on the panel.

### Case - Evaluation

1. The case is as small as reasonably possible given the output components used, and although larger than typical alarm clocks I think it's size is more than acceptable considering the features offered by it - not forgetting that it doubles up as a bedside lamp and fan, so space can be freed up by getting rid of any separate bedside lamp and/or fan!
2. All the components are contained within the case and are positioned very appropriately. However the power supply is external but this is not a problem at all.
3. The power supply is connected by a DC power jack located on the rear of the case.
4. The circuit board is firmly attached inside the case using stand-offs and positioned such that it is displayed as a design feature of the case.
5. There is a lamp override dimmer on the front panel. However when the alarm clock is used as a fan, the fan cannot be controlled independently of the lamp - that is, to have the fan on, the lamp must also be on. If the user wishes to sleep with the fan on but the lamp off this would be a problem. If I were to do an improved version of the alarm clock this would be one thing I would change. However rather than moving to using a PICAXE 28X (for the dual pwmout capable pins) I would continue using an 18X and a single MOSFET giving only one intensity setting - what I would do differently is that I would have both then fan and lamp controlled by relays, so although they would share intensity they could be switched on/off independently by the program - user control of this would be by means of the up/down buttons which would toggle the fan and lamp respectively (and they would both be overridden by the program during the wake-up call of course).

# Evaluation & Testing

---

## **General - Specification Points**

1. The project should be completed in less than 40 hours.

## **General - Evaluation**

1. While it is impossible to be sure how many hours I spent on the project it is obvious to me that it is more than 40 hours. My estimate is 120 hours and the breakdown of how I worked this out is shown in the diary section.

# Industrial Practices, Social Issues, Systems & Control

---

If my product were to be produced commercially, I would expect the PCBs to be made in small batches. The entire product would be split into smaller jobs for production. I would update the design and revert to the original aluminium construction of the side panels, because these could be produced more effectively when mass produced. The panels (both aluminium and acrylic) would all be cut, drilled and bent in batches with the aid of templates and drillings jigs to increase speed & accuracy. The use of jigs when drilling acrylic also serves to support the surrounding acrylic, which is otherwise at risk of cracking due to the pressure of drilling, so through the use of jigs the amount of wasted panels would be reduced.

The cut, drilled and bent panels would then be assembled on a production line with one person allocated to each stage of assembly, and the electronics then mounted in the cases.

I would update the main PCB to avoid using wire links for bridging tracks, as they are time consuming to place thus reducing profits. Instead I would use 0ohm resistors, as these are simply inserted with no prior wire stripping and cutting. If my product were to be produced on a larger scale still, they have the advantage that they can be handled by pick and place robots. Although I used 0ohm resistors where possible on my prototype, I had to use wire links in places because the gaps were too small for a 0ohm resistor to fit in. There are two possible solutions to this problem;

- Dual layer PCBs to reduce the need for bridging tracks
- Vertically placed 0ohm resistors

Dual layer PCBs are most advantageous because making the PCB dual layered would also allow the size to be reduced thus allowing the case to be smaller.

From an environmental and business point of view, it makes sense to use copper islands on the artwork. This is because they reduce the amount of copper to be etched, thus reducing etching chemical bills and increasing profits as well as reducing harmful wastes in the form of strong acids and alkalis.

With regard to quality of the final product, I would build quality assurance into the production of the alarm clocks, by continuing with the techniques I used for the prototype – weaving holes for all flying components and heat-shrink on all bare joints. The use of templates and jigs as already described would reduce inaccuracies due to human error.

# Industrial Practices, Social Issues, Systems & Control

---

I would also conduct quality control. This would mean that all PCBs would be tested before being mounted in the cases and random samples of the completed products would be tested both for electrical functionality and defects in the casing (e.g. cracks in the acrylic panels).

If demand for the product were great enough I would expect production to include greater automation to take advantage of the economies of scale. The main advantages of automation (by the use of robots/CNC machines) are that robots/CNC machines are almost exclusively more accurate than human workers, thus reducing material wastage due to products that fail testing. The other big advantage is that unlike human workers they do not require paying, though there are still expenses.

Pick and place robots could be used to populate the boards, and CNC machines would be used for the cutting, drilling and bending of the aluminium and acrylic panels. I used CAD and CAM throughout the project – examples of CAD are Livewire used in the creation of schematics, PICAXE Programming Editor for the programming of the PICAXE, PCB Wizard for the artwork production, ProDESKTOP for the design of the case, and finally, Techsoft 2D design for the design of the front panel which I then used in the manufacture of the front and mid panels via the Roland CAMM2 milling machine.

Finally, there is the functionality itself. As I already explained in my evaluation there are a few changes I would make to the design:

1. There is a lamp override dimmer on the front panel. However when the alarm clock is used as a fan, the fan cannot be controlled independently of the lamp - that is, to have the fan on, the lamp must also be on. If the user wishes to sleep with the fan on but the lamp off this would be a problem. If I were to do an improved version of the alarm clock this would be one thing I would change. However rather than moving to using a PICAXE 28X (for the dual pwmout capable pins) I would continue using an 18X and a single MOSFET giving only one intensity setting - what I would do differently is that I would have both the fan and lamp controlled by relays, so although they would share intensity they could be switched on/off independently by the program - user control of this would be by means of the up/down buttons which would toggle the fan and lamp respectively (and they would both be overridden by the program during the wake-up call of course).
2. The piezo is not as loud as it needs to be and so I would endeavour to find a louder alternative.

My product is a prototype, and with further development could be produced on a commercial scale.

# Bibliography & Conclusion

---

## Resources used

I have included the Wikipedia articles on:

- Bitmasks
- Binary-coded decimal

in appendix A for further reading if it interests you and the level I have explained to in the programming sections is not sufficient for this interest. The locations of these articles are <http://en.wikipedia.org/wiki/Bitmask> and [http://en.wikipedia.org/wiki/Binary-coded\\_decimal](http://en.wikipedia.org/wiki/Binary-coded_decimal) respectively. I also included the “ASCII printable characters” table in appendix A, from the Wikipedia article on ASCII, at <http://en.wikipedia.org/wiki/Ascii> (I modified the formatting slightly for printing).

I found a thread on the PICAXE user forums at [http://www.rev-ed.co.uk/picaxe/forum/Topic.asp?topic\\_id=1080&forum\\_id=9&Topic\\_Title=DS1307%2BCorrection&forum\\_title=No+new+posts+please%21+4](http://www.rev-ed.co.uk/picaxe/forum/Topic.asp?topic_id=1080&forum_id=9&Topic_Title=DS1307%2BCorrection&forum_title=No+new+posts+please%21+4) which helped me to recognise that the values stored in the DS1307 register are in binary coded decimal format and provided an example of how to print them correctly to the LCD.

I have included numerous datasheets in appendix from both <http://www.rev-ed.co.uk> and <http://www.rapidonline.com/>.

## Conclusion

On the whole I have really enjoyed the project, in particular the programming stages. It has been a great learning experience - I knew nothing of nibbles, words or stored binary coded decimals before the project (I had used BCDs in an up/down counter built during the course, but not in the same way that I have in this project).

Writing the program was a significant challenge, due partly to the difficulty of manipulating BCDs (printing them and performing addition/subtraction on them) and partly due to the very small amount of memory I had available for the compiled program. The challenge of optimising the code to minimise memory usage was again something that I really enjoyed. I have a suspicion that I was pushing the limits of the BASIC language in terms of the features offered; for example, the ability to define functions would have been very useful.

## Appendix A (Datasheets, Sample Program Thread)

---

This appendix contains:

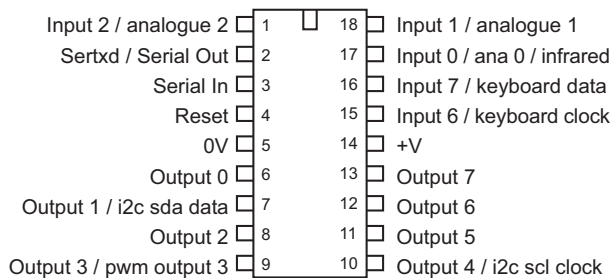
- PICAXE X Parts datasheet
- LCD AXE033 datasheet
- I2C Guide
- ASCII conversion table
- Sample Program Thread
- DS1307 datasheet
- Binary-coded decimals article
- Bitmasks article
- Backlit LCD module datasheet



# PICAXE 18X/28X/40X Extended Features...

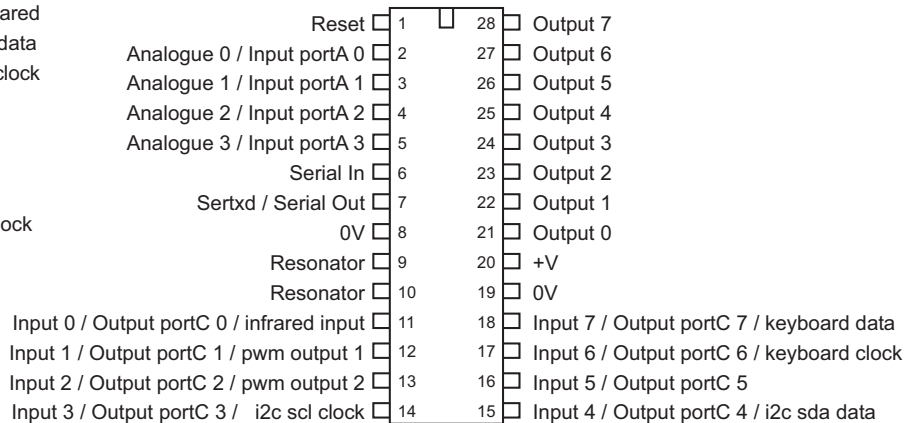
PICAXE Type	IC Size	Memory (lines)	I/O Pins	Outputs	Inputs	ADC (L =low)	Data Memory	Polled Interrupt
PICAXE-08	8	40	5	1-4	1-4	1L	128 - prog	-
PICAXE-18	18	40	13	8	5	3L	128 - prog	-
PICAXE-18A	18	80	13	8	5	3	256	Yes
PICAXE-18X	18	600	14	9	5	3	256 + i2c	Yes
PICAXE-28	28	80	20	8	8	4	64 + 256	-
PICAXE-28A	28	80	20	8	8	4	64 + 256	Yes
PICAXE-28X	28	600	21	9-17	0-12	0-4	128 + i2c	Yes
PICAXE-40X	40	600	32	9-17	8-20	3-7	128 + i2c	Yes

## PICAXE-18X



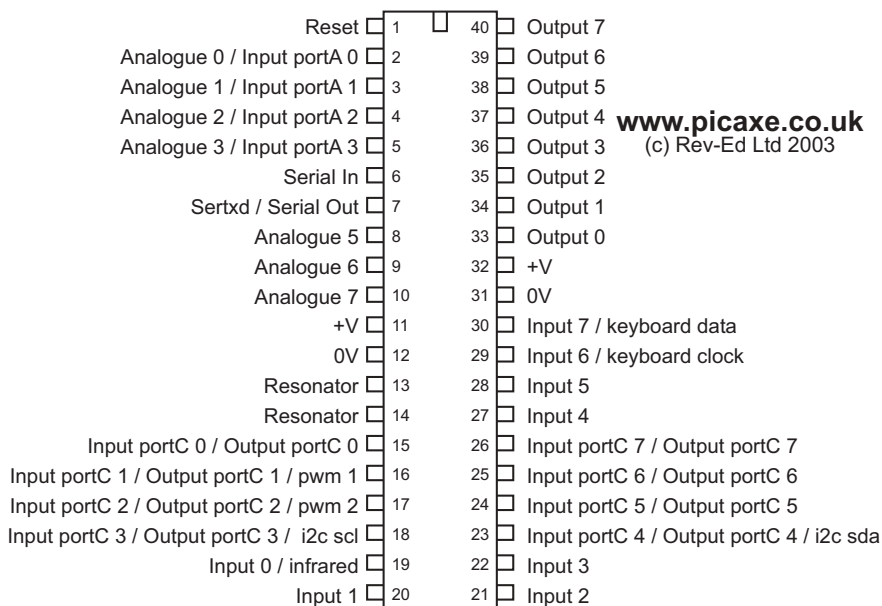
[www.picaxe.co.uk](http://www.picaxe.co.uk)  
(c) Rev-Ed Ltd 2003

## PICAXE-28X



[www.picaxe.co.uk](http://www.picaxe.co.uk)  
(c) Rev-Ed Ltd 2003

## PICAXE-40X



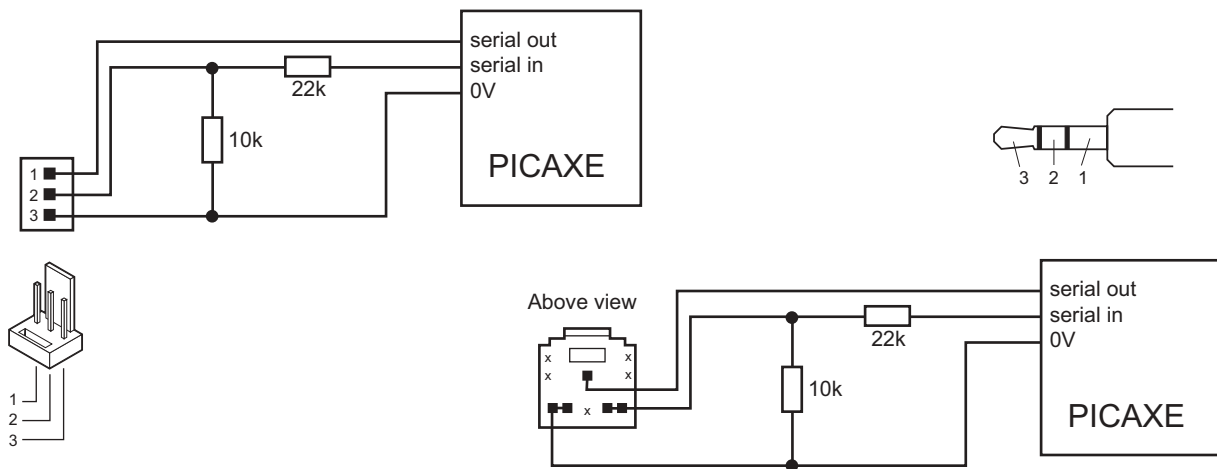
[www.picaxe.co.uk](http://www.picaxe.co.uk)  
(c) Rev-Ed Ltd 2003

### In this datasheet...

- Section 1 - PICAXE Commands
- Section 2 - What's New
- Section 3a - PICAXE-28X input/output pins
- Section 3b - PICAXE-40X input/output pins
- Section 4 - Resonator Frequency and Overclocking

## Serial Download Circuit:

The serial download circuit for all PICAXE microcontrollers is (straight or 'stereo plug' cable connections):



## SECTION 1 - PICAXE Commands: (new X part commands in bold)

Output -	high, low, toggle, pulsout, let pins =	Please see the BASIC Commands help file for more detailed syntax help and information about each command.
Sound -	sound	
Input -	if...then, readadc, <b>readac10</b> , pulsln, button	
Serial -	serin, serout, <b>sertxd</b>	
Program Flow -	goto, gosub, return, branch	
Loops -	for...next	
Mathematics -	let... (+, -, *, **, /, //, max, min, &,  , ^, &/,  /, ^/)	
Variables -	if...then, random, lookdown, lookup	
Data memory -	eeprom, write, read	
Delays -	pause, wait, nap, sleep, end	
Miscellaneous -	symbol, debug	
RAM -	peek, poke	
Servo Control -	servo	
Infrared -	infrain	
Interrupt -	<b>setint</b>	
Temperature -	<b>readtemp</b> , <b>readtemp12</b>	
Keyboard -	<b>keyin</b> , <b>keyled</b>	
1-wire Serial No -	<b>readownsn</b>	
I2C -	<b>readi2c</b> , <b>writei2c</b> , <b>i2cslave</b>	
PWM -	<b>pwmout</b>	
Counting -	<b>count</b>	

## SECTION 2 - What's new in the PICAXE-18X, 28X, 40X?

The extended X parts support all the standard commands and features, with the following enhancements:

- Program memory 8x as long (approx. 600 lines rather than 80), with intelligent download
- Continuously driven pwm motor drive outputs (pwmout command)
- Count high frequency pulses within a set time period (count command)
- Large data memory (128 or 256 bytes) (read/write commands)
- i2c bus support for EEPROMs and other devices (i2cslave/writei2c/readi2c commands)
- Interrupt feature on inputs (setint command)
- Accurate digital temperature sensor interface (readtemp/readtemp12 commands)
- 10 bit and 8 bit adc option (readadc10/readadc commands)
- User serial output via the serout pin / programming cable (sertxd command)
- 4800 baud rate option (and faster at higher clock frequencies) (serin/serout commands)
- Read serial number from any Dallas 1-wire device (e.g. iButton) (readownsn command)
- Computer keyboard interface on inputs 6 and 7 (keyin, keyed command)
- Software support for increased clock frequency (see section 3 of this datasheet).

See the BASIC Commands datasheet (v3.5 or greater) for further information on each command.

In addition the PICAXE-28X and 40X have a more **flexible i/o pin** layout to allow the user to select more inputs and/or outputs than the standard configuration. See section 2 of this datasheet.

### Memory Size

The X parts have a memory size 8x larger than the A parts (2048 bytes rather than 256 bytes). This means it can store a program of approximately **500-700 lines** of BASIC code (depending on commands used).

To reduce download times the X parts only download the appropriate (used) pages of memory. Therefore a shorter program will download quicker than a longer program

### PICAXE-40X

The PICAXE-40X is electronically configured as a 'special' version of the PICAXE-28X (with additional pins) Therefore when using the Programming Editor software the 'PICAXE-28X' mode is used for **programming both** the PICAXE-28X and the PICAXE-40X microcontrollers.

## SECTION 3a - PICAXE-28X Input/Output Pins

To provide greater flexibility, the input/output pin configuration of the PICAXE-28X can be varied by the user. The default power up settings are the same as the other PICAXE-28 parts (8 in, 8 out, 4 analogue).

**PORTA** (legs 2 to 5) provide 4 analogue inputs

(default) or up to 4 digital inputs.

**PORTB** (leg 21 to 28) provide 8 fixed outputs.

**PORTC** (leg 11 to 18) provide 8 digital inputs

(default) or up to 8 outputs.

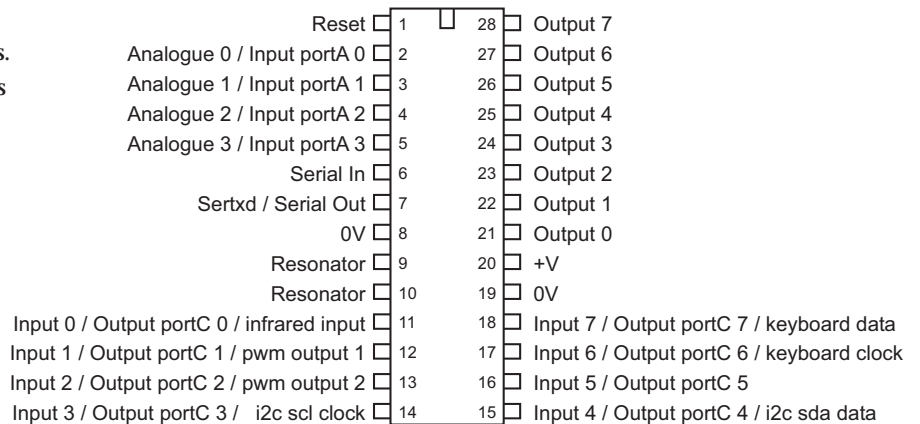
This gives a maximum of :

12 digital inputs

16 outputs

4 analogue inputs

### PICAXE-28X



### PORTA Functions

Leg	Default Function	Second Function
2	analogue 0	porta input 0
3	analogue 1	porta input 1
4	analogue 2	porta input 2
5	analogue	porta input 3

[www.picaxe.co.uk](http://www.picaxe.co.uk)  
(c) Rev-Ed Ltd 2003

### PORTB Functions

PORTB pins are fixed as outputs and cannot be altered.

### PORTC Functions

Leg	Default Function	Second Function	Special Function
11	input 0	output portc 0	infrared (input)
12	input 1	output portc 1	pwm 1 (output)
13	input 2	output portc 2	pwm 2 (output)
14	input 3	output portc 3	i2c scl clock (input)
15	input 4	output portc 4	i2c sda data (input)
16	input 5	output portc 5	
17	input 6	output portc 6	keyboard clock (input)
18	input 7	output portc 7	keyboard data (input)

The portC pins can be used as the default inputs, changed to outputs, or used with their special function via use of the infrain, keyin, i2cslave, or pwmout command as appropriate.

The second or special function of the pins are selected by modified commands as explained in the next section.

### SECTION 3b - PICAXE-40X Input/Output Pins

To provide greater flexibility, the input/output pin configuration of the PICAXE-40X can be varied by the user.

**PORTA** (legs 2 to 5) provide 4 analogue inputs (default) or up to 4 digital inputs.

**PORTB** (leg 32 to 40) provide 8 fixed outputs.

**PORTC** (leg 15-18, 23-26) provide 8 digital inputs (default) or up to 8 outputs.

**PORTD** (leg 19-22, 27-30) provide 8 digital inputs

**PORTE** (leg 8 to 10) provide 3 analogue inputs

This gives a maximum of :

- 20 digital inputs
- 16 outputs
- 7 analogue inputs

#### PORTA Functions

Leg	Default Function	Second Function
2	analogue 0	porta input 0
3	analogue 1	porta input 1
4	analogue 2	porta input 2
5	analogue	porta input 3

#### PORTB Functions

PORTB pins are fixed as outputs and cannot be altered.

#### PORTC Functions

Leg	Default Function	Second Function	Special Function
15	input portc 0	output portc 0	
16	input portc 1	output portc 1	pwm 1 (output)
17	input portc 2	output portc 2	pwm 2 (output)
18	input portc 3	output portc 3	i2c scl clock (input)
23	input portc 4	output portc 4	i2c sda data (input)
24	input portc 5	output portc 5	
25	input portc 6	output portc 6	
26	input portc 7	output portc 7	

The portC pins can be used as the default inputs, changed to outputs, or used with their special function via use of the i2cslave or pwmout command as appropriate.

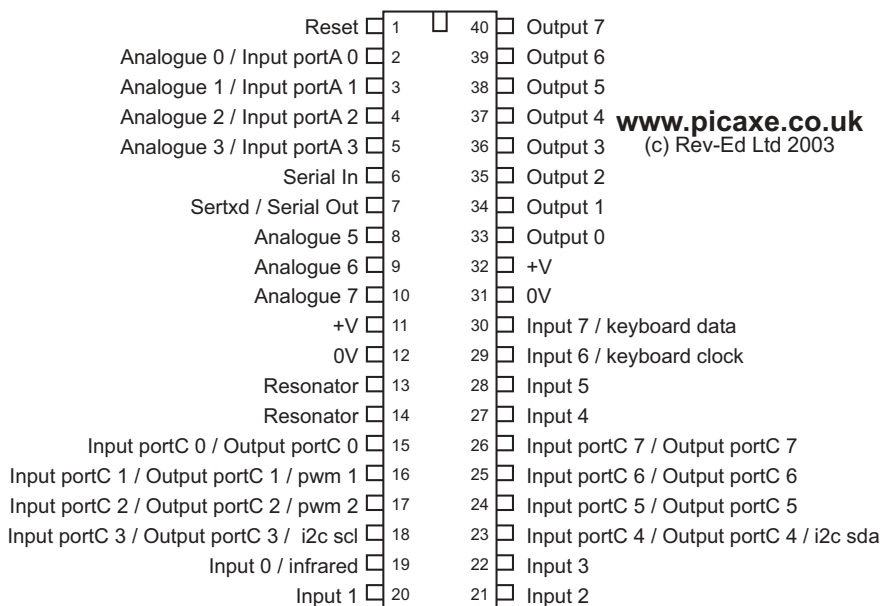
#### PORTD Functions

Leg	Default Function	Special Function
19	input 0	infrared (input)
20	input 1	
21	input 2	
22	input 3	
27	input 4	
28	input 5	
29	input 6	keyboard clock (input)
30	input 7	keyboard data (input)

#### PORTE Functions

PORTE pins are fixed as analogue inputs and cannot be altered.

#### PICAXE-40X



www.picaxe.co.uk  
(c) Rev-Ed Ltd 2003

### SECTION 3c - Using portA analogue inputs as digital inputs (28X, 40X)

The portA pins 0 to 3 (legs 2 to 5) are, by default, configured as analogue inputs. However with the PICAXE-28X and -40X they can also be used as simple digital inputs.

The following syntax is used to test the input condition:

```
if portA pin0 = 1 then jump
```

i.e. the additional keyword 'portA' is inserted after the 'if' command.

to test if two (or more) portA inputs are on

```
if portA pin0 = 1 AND pin1 = 1 then jump
```

to test if either of two (or more) portA inputs are on

```
if portA pin0 = 1 OR pin1 = 1 then jump
```

Note the portA command is only required once after the 'if' command.

It is not possible to test inputs on two different ports within the same if...then statement.

It is not possible to access the portA pins with any other 'input' type commands (count, pulsIn etc).

Therefore these pins should be reserved as simple on/off switches.

### SECTION 3d - Using portC as inputs (40X)

On the PICAXE-28X portC are the standard input pins and addressed by the standard `if pin0 =` command.

On the PICAXE-40X portD are the standard inputs, and hence use the standard `if pin0 =` command. Therefore for portC inputs the extra keyword portC must be used (as in the `if portA pin0 =` example above).

### SECTION 3e - Using portC as outputs (28X, 40X)

The portC pins are, by default, digital input pins.

However with the PICAXE-28X and -40X they can also be configured to be used as digital outputs.

To convert the pin to output and make it high

```
high portC 1
```

To convert the pin to output and make it low

```
low portC 1
```

To convert all the pins to outputs

```
let dirsc = %11111111
```

To convert all the pins to inputs

```
let dirsc = %00000000
```

Note that 'dirsc' uses the common BASIC notation 0 for input and 1 for output. (Advanced - If you are more familiar with assembler code programming you may prefer to use the command 'let trisc =' instead, as this uses the inverted assembler notation - 1 for input and 0 for output. Do not attempt to directly poke the trisc register (poke command) as the PICAXE bootstrap refreshes the register setting regularly).

To switch all the outputs on portC high

```
let pinsc = %11111111
```

```
(or) let portC = %11111111
```

To switch all the outputs on portC low

```
let pinsc = %00000000
```

```
(or) let portC = %00000000
```

To use portC 1 and portC 2 as pwm controlled outputs use the **pwmout** command (see the BASIC Commands help file for further information). The pwm output is maintained continuously in the background, making these pins ideal for controlling motors etc.

It is not possible to access the portc pins with any other 'output' type commands (sound, serout, pulsout etc). Therefore these pins should be reserved as simple on/off outputs (apart from the pwm control on 1 and 2).

When using the special input functions (infrared sensor (0), or an i2c device (3, 4), or a keyboard (6, 7) ) you must take care to ensure that the appropriate pins are maintained as inputs. Converting these pins to outputs may damage the external device and/or the microcontroller.

## SECTION 4 - Resonator Frequency and Overclocking.

All PICAXE functions are based upon a 4MHz resonator frequency. This is the only frequency recommended. However the user may choose to 'overclock' the X parts if desired, although this is not recommended unless absolutely necessary for a particular project (e.g. when using the count command).

With the -08, -18, -18A the internal resonator is fixed at 4MHz and cannot be altered.

With the -18X the internal resonator has a default value of 4MHz. However it can be increased by the user to 8MHz via use of the 'setfreq' command.

With the -28 and -28A an external 4MHz resonator must be used.

With the -28X / -40X an external 4MHz 3 pin ceramic resonator is normally used, but it is also possible to use a faster resonator (8 or 16Mhz), although this will affect the operation of some of the commands.

NB PICAXE-28X firmware version 7.0 can be used at 4 or 8 MHz

PICAXE-28X or -40X firmware version 7.1 (or greater) can be used at 4, 8 or 16 MHz

The Programming Editor software supports resonator frequencies of 4, 8 and 16MHz only. No other frequencies are supported. If any other frequency is used it will not be possible to download a new program into the PICAXE microcontroller.

*To change the frequency:*

### PICAXE-18X

Download a program containing the command `setfreq m4` (for 4 MHz) or `setfreq m8` (for 8Mhz). If no `setfreq` command is used in a program the frequency will default to 4MHz. Note the new frequency occurs immediately after the command is run. When downloading new programs, you must ensure the correct frequency (View>Options>Mode) is used to match the last program running in the PICAXE-18X chip. If in doubt perform a 'hard-reset' at 4Hz.

### PICAXE-28X and PICAXE-40X

Solder the appropriate external 3pin ceramic resonator into the project board.

### Downloading programs at 4, 8, 16MHz

After changing frequency you must select the correct frequency via the View>Options>Mode software menu. If the wrong frequency is selected the program will not download.

### Commands affected by resonator frequency.

Many of the commands are affected by a change in resonator frequency. A summary of the important commands affected are given below (see BASIC Commands datasheet for detailed command syntax).



**count**

The base unit of count is

1ms at 4MHz  
0.5ms at 8 MHz  
0.25ms at 16 MHz

The pin is checked every

20us at 4MHz (max. 25kHz pulse rate)  
10us at 8MHz (max. 50kHz pulse rate)  
5us at 16MHz (max. 100kHz pulse rate)

**i2slave**

The bus speed within i2slave must be adjusted by use of the appropriate frequency keyword

i2cfast / i2cslow at 4Mz  
i2cfast8 / i2cslow8 at 8Mz  
i2cfast16 / i2cslow16 at 16Mz

If the incorrect keyword is used the i2c function may not work.

**pause / wait**

The base unit of pause is:

1ms at 4MHz  
0.5ms at 8 MHz  
0.25ms at 16 MHz

The base unit of wait is:

1s at 4MHz  
0.5s at 8 MHz  
0.25s at 16 MHz

**pulsout / pulsln**

The base unit of pulsout/pulsln is:

10us at 4Mhz  
5us at 8Mhz  
2.5us at 16Mhz

**pwmout**

The period and duty cycle should be calculated using 4MHz, 8MHz or 16Mhz as appropriate.

**serin / serout / sertxd**

Due to the sensitive nature of serial communication no guarantee is given that serin or serout commands will work at any frequency other than 4MHz. However the theoretical baud rates at the higher clock frequencies are as follows:

Baudmode	4MHz	8MHz	16MHz
600	600	1200	2400
1200	1200	2400	4800
2400	2400	4800	9600
4800	4800	9600	19200 (also sertxd baud rate)

A maximum of 4800 is recommended for complicated serial transactions.

**sound**

The note of sound will be multiplied by 2 (8Mhz) or 4 (16MHz).

The duration of sound (12ms at 4MHz) will be reduced to 6ms (8MHz) or 3ms (16MHz)

**Commands that do not work at 8 or 16MHz**

The following commands will not work at 8 or 16MHz due to timing issues with the external device listed:

- infrain (infrared remote)
- keyin (keyboard)
- keyed (keyboard)
- readtemp / readtemp12 (DS18B20 temperature sensor)
- readown (1-wire device)
- servo (servo)

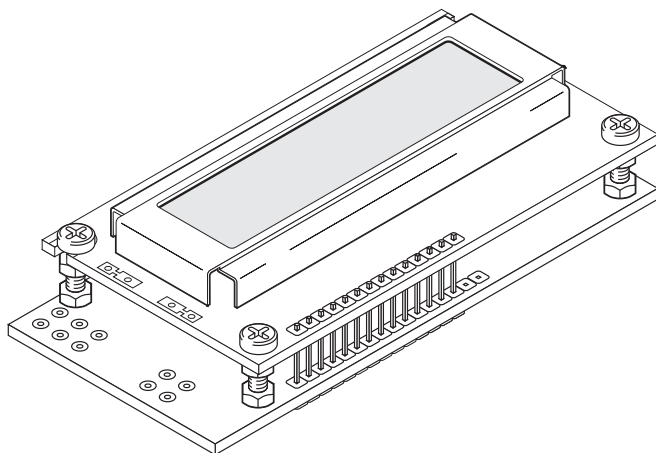
**Commands that are not affected by frequency changes.**

The following timing commands are NOT affected as they use a separate internal r/c timer:

- nap and sleep

# SERIAL/I2C LCD AND CLOCK (V2)

---



The serial LCD and clock module allows microcontroller systems (e.g. PICAXE) to visually output user instructions or readings, without the need for a computer. This is especially useful when working, for example, with analogue sensors, as the analogue reading can easily be displayed on the LCD module. All LCD commands are transmitted serially via a single microcontroller pin using the `serout` command. e.g.

to print the text 'Hello' the command is simply:

```
serout 7,N2400,("Hello")
```

The module can also store 7 programmable pre-defined messages to save memory space usage within the PICAXE system.

The optional low-cost clock upgrade provides a real-time clock and programmable alarm output. The LCD can show the current date and time on it's display, and the alarm output can be programmed to trigger at any period between 10 seconds and 1 year. The clock has a lithium coin cell backup that maintains the time for up to ten years when the main power supply is removed.

## Key Features:

1. 16x2 LCD Alphanumeric Display
2. Simple serial (1 wire) connection to microcontroller (2400,N,8,1).
3. Optional i2c interface to PICAXE-X parts.
4. 7 Programmable pre-defined messages
5. Small footprint (almost same size as the LCD).
6. Optional low-cost clock upgrade, providing
  - Real Time Clock
  - Programmable Alarm Output
  - 1Hz pulse output
  - 10 year battery backup

## Which Mode? (serial or i2c)

Most users will use the module in the default serial mode. The only reason to use it in i2c mode is if:

- 1) You are using a PICAXE-X chip **and**
- 2) You wish to read the time/data from the DS1307 clock upgrade directly into the PICAXE.

In all other cases the serial mode should be used.

In i2c mode the LCD module acts as a 'dumb' i2c slave device. The clock and alarm functions are not available - all clock and alarm functions must be carried out by the PICAXE X part itself.

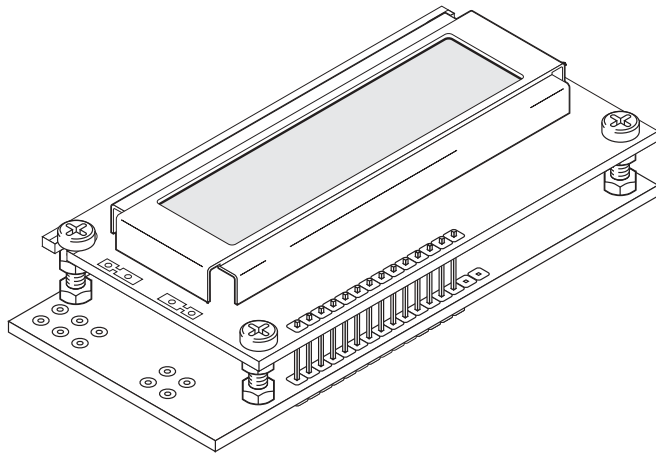
See Section 1 for construction and assembly details.

See Section 2 on pages 6-7 for i2c connection details and samples. Further information about i2c protocol and interfacing can be found in the 'i2c Tutorial' help file. It is assumed that the user has already read this help file.

See Section 3 on pages 8-15 for serial connection details and samples.

## Note: Version 1 LCD modules

This datasheet is for version 2 (black colour) modules. Version 1 (green colour) modules did not have the i2c mode, but this datasheet can still be used as the serial mode information in section 3 also applies to version 1 modules.



## Section 1 - Construction and Kit Contents

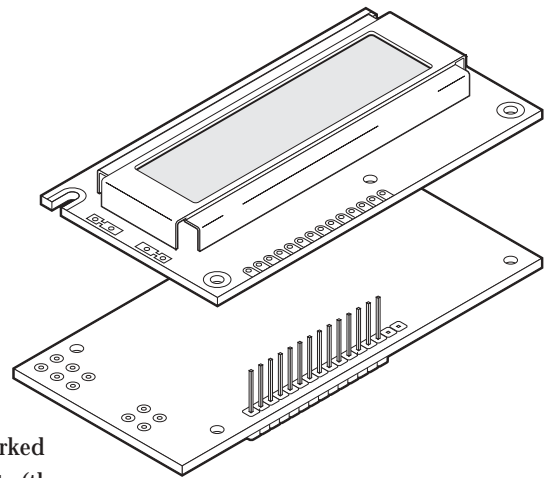
- pre-populated PCB
- 16x2 alphanumeric display (brand may vary)
- bag of 12 nuts, 4 bolts and 3 support headers

The LCD is supplied loose so that it can be either fitted directly to the board, or connected via a longer ribbon cable connection if desired. The following instructions explain how to fit the LCD directly to the board (track side) and presume the user is confident at soldering.

### Connecting the LCD

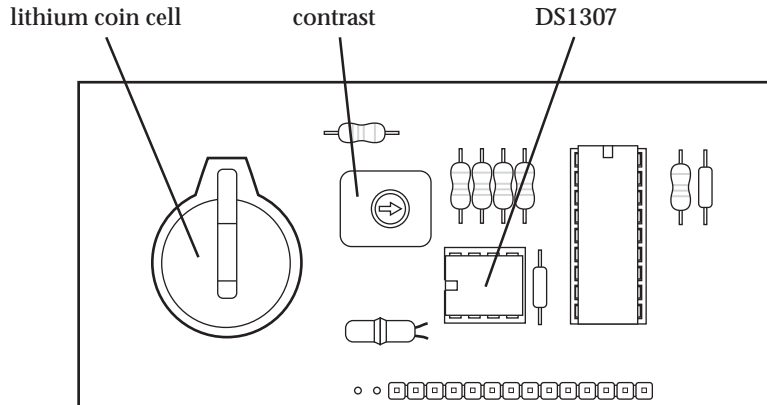
1. Snap one header into a 4 and 6 way section. Place the short end of the 4 way section, with another 10 way header, through the holes labelled 1-14 on the pcb (from the track side, so that the short ends show on the component side). Note the extra holes marked A and K and resistor RB are only used on LCDs with LED backlights (the LCD in the kit does not have a backlight and so these connections are not required).
2. Carefully solder each wire pin on the component side of the pcb. Check each joint carefully for shorts between pins.
3. Place the four bolts through the LCD (from the top downwards). Loosely fit two nuts to each bolt to act as a spacer.
4. Slide the LCD onto the headers on the track side of the pcb, carefully aligning the headers and bolts. When aligned, carefully tighten the spacer nuts so that the LCD is lying parallel to the pcb. Add another nut to each bolt to hold the LCD in position.
5. Solder the LCD to the pin headers.
6. Snap the 6 pin header into a 2 and 4 way section. Solder the two way section to the CLK contacts on the board.
7. Solder a wire link in position J2 (power) if a 4.5V battery pack is to be used. This is not required for a 5V or 6V supply.
8. Connect a power supply to the main connection header (red wire to V+, black wire to 0V). The LCD should display a time message **when the two CLK contacts are shorted** (e.g. with the jumper provided in the kit) and once **the contrast is adjusted** (via the variable resistor marked 'contrast'). If the LCD does not display a message check the power, contrast and the 14 connector pins carefully. (Note that if the optional clock upgrade chip is not fitted, the time will always show as 00/00/00 00:00)
9. Solder a wire link in position J1 (mode) if the LCD is to be used in i2c mode. No link is required for the default serial mode.

See page 8 for a sample PICAXE serial test program.



Note that the LCD is fitted above the TRACK side of the PCB. Ensure no solder bridges between pins on the header.

## Installing the Optional Clock Update



### Required:

- CR2032 lithium coin cell
- DS1307 Clock IC

### Instructions:

1. Place the DS1307 into the 8 pin socket, ensuring pin 1 is facing the lithium cell holder.
2. Place the CR2032 lithium coin cell in the holder, ensuring the positive (+) side is facing up.

### Notes:

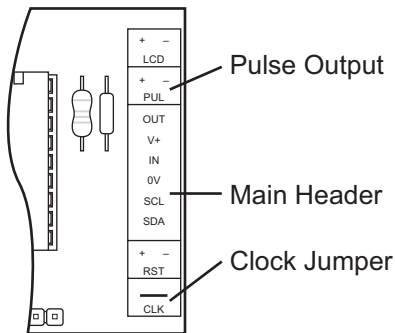
Note that the lithium coin cell keeps the DS1307 clock operating when the main power supply is not connected. This ensures accurate time is kept by the module. The coin cell **does not** power the LCD or the pulse output. The coin cell will last approximately 10 years.

Note that the clock and alarms (and pulse output) will not operate correctly until the initial time is programmed into the module (see the 'Programming Time into the Module' section below).

### Users in Europe/USA

Please note that the date convention used in the module is the UK date format dd/mm/yy. The US date format mm/dd/yy is available by special order.

## Input / Output / Power Connections



### Main Header (V+,0V)

The main header provides connection for the power supply (5-6V DC on V+). If you wish to use 4.5V solder a wire link in position J2 (power). This shorts out the voltage protection diode D1, as this diode causes a 0.7V voltage drop, which can make the screen very dim at this lower 4.5V voltage.

### Main header (IN)

These is the serial input (IN).

### Main header (SDA and SCL)

These are the i2c mode connections. They must only be used when a wire link has been soldered in position J1 to put the module into i2c mode (see section 2).

### Main header (OUT)

The alarm output triggers (goes high for 5 seconds) whenever a clock alarm occurs (in serial mode). The alarm output can sink or source 20mA.

### Pulse Output (PLS)

The pulse output outputs a square wave of 1Hz (1 pulse per second) when the optional DS1307 clock IC is fitted. A 330R resistor is included on the board to allow a low current LED to be soldered directly to this connection to provide a flashing 'second' indicator. The pulse output can sink or source 20mA. The pulse output will not operate until the clock upgrade is fitted and the correct time is programmed into the unit.

### Clock Jumper (CLK)

When the clock jumper is fitted the module goes into clock mode. During this mode instructions cannot be sent via the serial connection, as the unit is acting as a standalone 'alarm clock'. User defined message 1 is constantly shown on the top line of the LCD and the time is constantly shown on the bottom line of the LCD (when the module is powered). The pulse output and alarm output operate as normal.

### LCD Backlight (LCD) (the LCD provided in the kit does not have a backlight)

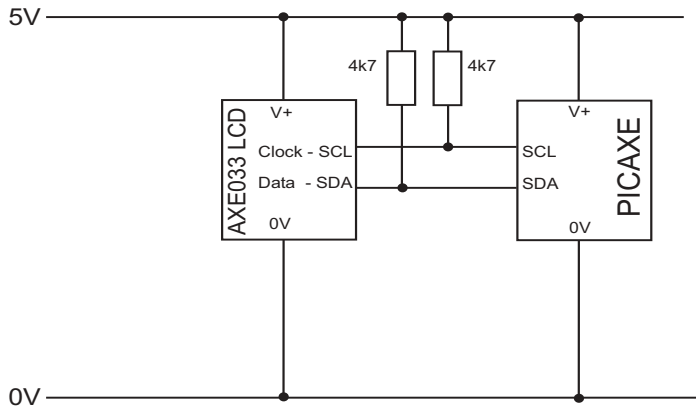
When a backlight is fitted the LCD connections allows power to be applied to the backlight. Note that you must also have soldered the 'A' and 'K' connections on the LCD and added a resistor to position RB on the pcb to use this feature. See your LCD datasheet for suitable power and resistor values.

### Resonator Tuning (RST)

The resonator tuning pin allows the internal resonator to be calibrated for use with various PICAXE chips when in serial mode. See appendix a, resonator tuning, for more details.

## Section 2) Connecting the Module to a PICAXE Microcontroller (i2c mode)

The following diagram shows how to connect the LCD module to the PICAXE X i2c system.



Note that the 4k7 resistors pull up resistors are pre-soldered onto the AXE033 LCD module. Therefore additional external resistors are not required.

### i2c programming details

The i2c communication protocol used with the LCD module is the same as popular eeprom's such as the 24C04. The SPE030 family code is \$C6, operates at slow speed (i2cslow) and has a single byte (i2cbyte) address size. Therefore the PICAXE i2c setup command (required before readi2c or writei2c is used) is  
i2cslave \$C6,i2cslow,i2cbyte

#### PICAXE (i2c) Test program

```
init: pause 500           \ wait for display to initialise
      i2cslave $C6,i2cslow,i2cbyte \ set up i2cslave for LCD
main: writei2c 0,(254,128,255) \ move to start of first line
      pause 10             \ wait for LCD to process data
      writei2c 0,("Hello!123",255) \ output text
      end
```

The display is write only in i2c mode. Do not use the readi2c command at slave address \$C6, as may cause unreliable behaviour which will require the module to be reset. Note that a 10ms delay (pause 10) should be placed between consecutive writei2c commands to allow time for the data to be processed.

The LCD can display characters and can also accept certain control commands (e.g. clear display or move cursor to new position). Note that the LCD module takes approx half a second to initialise and so any data sent during this period will be lost. It is advisable to put a 'pause 500' command at the start of any program to ensure no data is lost when the system is powered up.

Characters are normal symbols that can be displayed on the LCD screen. See Appendix 1 for a table of the common ASCII characters.

All LCD data is written to the write buffer at address 0. This buffer stores the data, and then prints the data on the LCD screen at the current cursor position when the special byte '255' is received. The buffer has a maximum size of 20 characters. Each write must terminate with the number 255, as this tells the module to start writing the buffered characters to the LCD display itself. Allow 10ms for this processing.



### Control Commands (254)

All LCD commands (move cursor etc) are preceded by the number 254.

The most common control commands are

<code>writei2c 0,(254,1,255)</code>	Clear Display (must be followed by 'pause 30')
<code>writei2c 0,(254,8,255)</code>	Hide Display
<code>writei2c 0,(254,12,255)</code>	Restore Display
<code>writei2c 0,(254,14,255)</code>	Turn on Cursor
<code>writei2c 0,(254,16,255)</code>	Move Cursor Left
<code>writei2c 0,(254,20,255)</code>	Move Cursor Right
<code>writei2c 0,(254,128,255)</code>	Move to line 1, position 1
<code>writei2c 0,(254,y,255)</code>	Move to line 1, position x (where $y = 128 + x$ )
<code>writei2c 0,(254,192,255)</code>	Move to line 2, position 1
<code>writei2c 0,(254,y,255)</code>	Move to line 2, position x (where $y = 192 + x$ )

### Using the Optional Clock Upgrade in i2c mode

When the clock upgrade is used the PICAXE must read the data directly from the DS1307 chip and then issue LCD write commands to display the data on the screen. The LCD module has no internal 'intelligent' clock routines when in i2c mode, as only the PICAXE can access the data (the LCD module is a slave, not master, device). Remember that when reading/sending the data to both LCD and DS1307 it is necessary to **keep re-issuing** the appropriate i2cslave command for each part.

### Setting the Time / Date

To set the correct time after the circuit is first powered up, the current time must be written to the DS1307 registers. The following example PICAXE program will setup the time to 11:59:00 on Thursday 25/12/03. This is carried out by loading the registers in order from address 00 upwards i.e. seconds then minutes then hours etc.

```
i2cslave %11010000, i2cslow, i2cbyte
writei2c 0, ($00, $59, $11, $03, $25, $12, $03, $10)
end
```

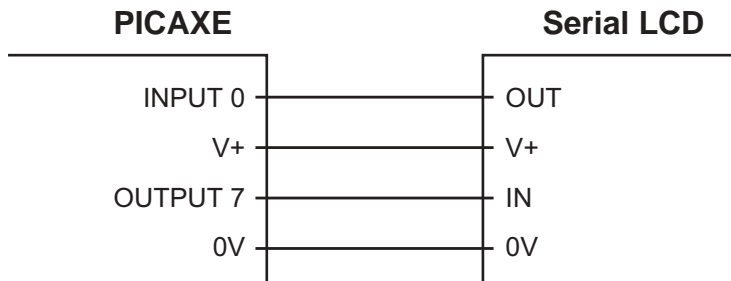
### Reading the Time / Date

To read the current time you can use the following program to load variables within the PICAXE with the various register values from the DS1307. This example program acts as an alarm clock, checking the time every 30 seconds. If the time is exactly 07:00 then a buzzer, connected to output 7, will sound for 20 seconds.

```
i2cslave %11010000, i2cslow, i2cbyte      ` set slave details
loop:
    pause 30000                          ` wait 30 sec
    readi2c 0, (b0, b1, b2)              ` read sec, min, hour
    if b2 <> $07 then loop                ` if hour not 7 loop
    if b1 <> $00 then loop                ` if min not 00 loop
    high 7                                ` switch on buzzer
    pause 20000                           ` wait 20 sec
    low 7                                  ` switch off buzzer
    pause 60000                           ` wait 60 sec
    goto loop                              ` loop
```

### Section 3 - Connecting the LCD to a PICAXE Microcontroller (serial mode)

The following diagram shows how to connect the Serial LCD module to the PICAXE system. Output 7 is used to send signals to the LCD, and input 0 is used for the alarm signal from the clock alarm.



Note: The connections must be made **directly** to the PICAXE output pins (not via the darlington driver buffered outputs found on the PICAXE starter pack project boards)

#### PICAXE Test program

```
init: pause 500           \ wait for display to initialise
main: serout 7,N2400,(254,128) \ move to start of first line
      serout 7,N2400,("Hello!123") \ output text
      end
```

Note the use of N2400 (not T2400) within PICAXE programs. Note the Serial LCD does not buffer bytes received, and so a small delay between bytes (to update the display) is required on non-PICAXE systems. This delay is applied automatically by the PICAXE system.

If the characters do not all appear as expected (e.g. as non standard symbols), see the 'Resonator Tuning' section of the datasheet.

#### Displaying Messages

The LCD can display characters, messages and the time, and can also accept certain control commands (e.g. clear display or move cursor to new position). Note that the serial LCD module takes approx half a second to initialise and so any data sent during this period will be lost. It is advisable to put a 'pause 500' command at the start of any program to ensure no data is lost when the system is powered up.

#### Characters

Characters are normal symbols that can be displayed on the LCD screen. See Appendix 1 for a table of the common ASCII characters. Note that 0-7 are special characters that actually print out the time and predefined messages. The numbers 253 and 254 are used to indicate a write memory or control command sequence follows.

0	Time
1-7	Predefined Messages
8-128	ASCII Characters (see Appendix 1)
129-252	Miscellaneous Characters (may vary dependant on LCD type)
253	Special Command – Write Memory
254	Special Command – Command Character
255	Reserved for future use

Characters can be output via two methods – either by using the ASCII number or the symbol enclosed in speech marks e.g. (65) and (“A”) both output the same symbol.

#### Control Commands (254)

Control commands are all prefixed by the number 254. They are used to send commands to the Serial LCD Module (e.g. move to line 2, switch cursor off etc.).

The most common control commands are

254,1	Clear Display (must be followed by a ‘pause 30’ command)
254,8	Hide Display
254,12	Restore Display
254,14	Turn on Cursor
254,16	Move Cursor Left
254,20	Move Cursor Right
254,128	Move to line 1, position 1
254, y	Move to line 1, position x (where $y = 128 + x$ )
254,192	Move to line 2, position 1
254, y	Move to line 2, position x (where $y = 192 + x$ )

#### Write Commands (253)

Write commands are all prefixed by the number 253. They are used to program the predefined messages, current time or alarm times into the Serial LCD module.

0	Set clock time
1-7	Set predefined messages 1-7
8	Set Alarm (date/time)
9	Set Alarm (interval)
10	Turn Alarm Off

All write commands must be followed by a 1000ms delay (pause 1000 command) to allow the internal save to be carried out. When a write command is used a brief ‘DATA SET’ message will appear on the top line of the LCD to indicate the data has been saved. See the sections below for more details.

## Programming a Predefined Message into the Module

The module can contain 7 user predefined messages, each message 16 characters long. These messages are stored on the LCD module and can be used to greatly reduce the 'display text' that must be stored within the PICAXE or Stamp (hence reducing the length of the program).

Messages 1,3,5,7 automatically appear on the top line of the display.

Messages 2,4,6 automatically appear on the second line of the display.

The messages must be programmed into the module using a small program running in a microcontroller such as the PICAXE. The following instructions presume the connection as shown in the example PICAXE circuit above.

To set message 1 to "Player 1 =" and message 2 to "Player 2 =" program the PICAXE with the following program. This loads the message write instruction (253), followed by the message memory address (1 or 2) followed by the message.

```
init:      pause 500
main:      serout 7,N2400,(253,1,"Player 1=    ")
           pause 1000
           serout 7,N2400,(253,2,"Player 2=    ")
           pause 1000
           end
```

Note the messages must always be 16 characters long, so additional spaces **must be** added to the text to ensure the message is **exactly 16 characters long**. Note that a 1000 millisecond programming period must be added after every write instruction.

## Displaying a Predefined Message

The predefined displayed messages are displayed in the same way as normal characters, using the character code 0 (time) or 1 to 7 (messages). Note that a 10ms delay (pause 10 command) must be added after each command to give the LCD enough time to display all the 16 characters in the message.

Therefore the following program will display message 1 on the top line of the display, and the time on the bottom line of the display.

```
init:      pause 500
main:      serout 7,N2400, (1)
           pause 10
           serout 7,N2400, (0)
           pause 500
           goto main
```

## Combining Predefined Messages and Variables

It is often useful to combine predefined messages with variables e.g. displaying the score of a game. The following program shows how to show the two scores from two players, presuming message 1 and message 2 have been pre-programmed with the phrases "Player 1=" and "Player 2=" (see above).

```
init:      pause 500
main:      serout 7,N2400,(1)
           pause 10
           serout 7,N2400,(254,137,#b1," ")
           serout 7,N2400,(2)
           pause 10
           serout 7,N2400,(254,201,#b2," ")
           let b1 = b1 + 1
           let b2 = b2 + 2
           pause 500
           goto main
```

Note that the message code (1 or 2) is first output. A delay of 10ms is then added to allow the LCD module to display the message. The cursor is then moved 9 positions along the screen (to the position after the = sign by the 254,137 or 254,210 command) and then the variable value is displayed. Note that the # symbol tells the microcontroller to output the ASCII equivalent of the variable value, not the direct value (e.g. "6" "5" not the value 65, which would actually appear as the character "A"!)

Two additional spaces are then also added to ensure variable value changes are overwritten correctly (e.g. to overwrite '234' by '1' you must output '1(space)(space)' to ensure the '34' of the first number is overwritten by the spaces.)

## Programming the Time into the Module

The current time must be programmed into the module using a small program running in a microcontroller such as the PICAXE. The following instructions presume the connection as shown in the example PICAXE circuit above. Note that once set, the lithium coin cell will maintain the clock time accurately for up to ten years.

To set the clock to 22:00 on 25/12/01 program the PICAXE with the following program. This program loads the write instruction (253), followed by the clock memory address (0), followed by the date and time ("25/12/01 22:00 ")

```
init:      pause 500
main:      serout 7,N2400, (253,0,"25/12/01 22:00 ")
           pause 1000
           serout 7,N2400, (0)
           end
```

Note the time and date must be presented exactly as shown, using the 24 hour clock format dd/mm/yy hh:mm. Note the write messages must always be 16 characters long, so 2 spaces are added to the text to ensure the message is exactly 16 characters long. Note that a 1000 millisecond programming period must be added after every write instruction. The last serout command shows the time to check it is correctly programmed.

To accurately enter a time, download the program (set with a time about 1 minute ahead of schedule) into the PICAXE. Then press the reset switch on the PICAXE (to re-run the program) at exactly the correct time. This will set the time accurately.

## Displaying the Time

The time message is updated with the current date/time every time it is used. The time message is displayed in the same way as normal preset messages, using the special character code 0. The time always automatically appears on the second line of the display.

Therefore the following program will display message 1 on the top line of the display and the time on the bottom line of the display. The screen will update the time every 0.5 second.

```
init:      pause 500
main:      serout 7,N2400, (1)
           pause 10
           serout 7,N2400, (0)
           pause 490
           goto main
```

## Programming the Alarm Time into the Module

The alarm output pin triggers (goes 'high' for 5 seconds) whenever the alarm time is reached. The alarm can be set to a specific date/time (write code 8), or can be set to repeat at certain time intervals (write code 9). Only one alarm type is active at any time – the last written alarm type is the one used within the module.

The alarm time or interval must be programmed into the module using a small program running in a microcontroller such as the PICAXE. The following instructions presume the connection as shown in the example PICAXE circuit above.

### Setting an alarm at a specific time:

To set the alarm time clock to 07:30 every day (using write code 8), program the PICAXE with the following program. This program loads the write instruction (253), followed by the alarm address (8), followed by the time ("00/00/00 07:30")

```
init:      pause 500
main:      serout 7,N2400, (253,8,"00/00/00 07:30 ")
           pause 1000
           end
```

Note the time and date must be presented exactly as shown, using the 24 hour clock. The '00' characters can be used (within the date only) to indicate an 'ignore' condition, so in the example above the date is completely ignored, so the alarm will trigger every day at 07:30. Note the write messages must always be 16 characters long, so 2 spaces are added to the text to ensure the message is exactly 16 characters long. Note that a 1000 millisecond programming period must be added after every write instruction.

To set the alarm to trigger on the first of every month at midnight

```
init:      pause 500
main:      serout 7,N2400, (253,8,"01/00/00 00:00 ")
           pause 500
           end
```



**Setting an alarm at a specific time interval:**

To set the alarm to trigger at an interval, instead of an exact time, use write code 9 instead of 8. For example, to trigger the alarm every ten minutes (using write code 9)

```
init:      pause 500
main:      serout 7,N2400, (253,9,"00:10:00    ")
           pause 1000
           end
```

To set the alarm to trigger every 30 seconds

```
init:      pause 500
main:      serout 7,N2400, (253,9,"00:00:30    ")
           pause 1000
           end
```

To set the alarm to trigger every two hours

```
init:      pause 500
main:      serout 7,N2400, (253,9,"02:00:00    ")
           pause 1000
           end
```

Note the alarm trigger interval is denoted by a number of hours (00 to 23), minutes (00 to 59) and seconds (00 to 59) between alarms. The smallest practical alarm interval is 10 seconds, due to the five second 'on time' of the alarm output. Note the write messages must always be 16 characters long, so 8 spaces are added to the text to ensure the message is exactly 16 characters long. Trigger values longer than one day should be set using the time and date method shown above.

**IMPORTANT NOTE**

The interval timer operates as follows on power-up:

Upon power up the module reads the current time – and then adds the alarm interval to the current time to generate the next alarm time. When an alarm occurs the interval is once again added to the current time to create the next alarm time.

Therefore the interval timer is effectively reset every time the module is powered down – the first alarm will be activated the 'interval time' after power up. To keep the interval exactly consistent over a long period the module must be continuously powered.

**Turning the alarm off:**

To disable either type of alarm send the '10' command (note that the 10 command does not require 16 characters to be sent as with all the other commands – it is just sent by itself)

```
init:      pause 500
main:      serout 7,N2400, (253,10)
           pause 500
           end
```

# APPENDIX A

## Internal Resonator Tuning

The microcontroller used as the controller on the serial LCD operates from an internal resonator. Many of the PICAXE chips also operate using the internal resonator.

Use of the internal resonator reduces cost of the product and simplifies PCB design. In most cases this causes no problems and the LCD will function correctly without any calibration.

However the internal resonator is not as accurate as external crystal devices, and it has been found on a very small number of PIC16F628 that the calibration of the internal resonator can drift slightly. If the PICAXE internal resonator frequency is also at one of the calibration extremes, you may at first experience some 'corrupt' characters being displayed on screen. Typically numeric characters will work correctly, but text may appear as non-standard symbols.

If you experience this issue, simply solder a wire link across the 'RST' pads. This will adjust the resonator frequency to allow correct operation.

### Earlier Firmware (Version 1 (green) PCB)

Note the RST pin was used as a reset on earlier firmware releases. To discover if you have the resonator calibration feature, try three tests:

- 1) If this datasheet is supplied within the LCD pack it is automatically a latest firmware edition. The resonator tuning feature can be used immediately.

*If this datasheet has been downloaded from the internet:*

- 2) Power up the LCD with the clock jumper in place. If the display shows 'Serial LCD&Clock' it is the new firmware. If the '&' symbol is displayed as a '/' symbol it is the older firmware.
- 3) Simply make the wire link. If the LCD does not function then you require a free firmware upgrade.

If you require a **free** upgrade of earlier firmware, simply return the PIC chip, suitably packaged with proof of purchase (e.g. copy of delivery note or invoice) to your regional distributor who will re-program the firmware chip and return it to you free of charge.

Europe - Revolution Education Ltd ([www.picaxe.co.uk](http://www.picaxe.co.uk))

# APPENDIX B

Standard Character Pattern (Elec & Eltek LCD Module)

Upper(4bit)		Lower(4bit)	LLLL	LLHL	LLHH	LHLL	LHLH	LHHL	LHHH	HLLL	HLLH	HLHL	HLHH	HHLL	HHLH	HHHL	HHHH
LLLL	CG RAM (1)			0	1	2	3	4	5	6	7	8	9	A	B	C	D
LLLH	(2)		!	1	Q	a	9										
LLHL	(3)		"	2	R	b	r										
LLHH	(4)		#	3	S	s											
LHLL	(5)		\$	4	T	t											
LHLH	(6)		%	5	U	u											
LHHL	(7)		&	6	V	v											
LHHH	(8)		'	7	W	w											
HLLL	(1)		(	8	X	x											
HLLH	(2)		)	9	Y	y											
HLHL	(3)		*	:	J	j											
HLHH	(4)		+	;	K	k											
HHLL	(5)		,	<	L	l											
HHLH	(6)		-	=	M	m											
HHHL	(7)		.	>	N	n											
HHHH	(8)		/	?	O	o											

Standard Character Pattern (Powertip LCD Module)

		Higher 4-bit (D4 to Character Code (Hexadecimal))																		
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
Lower 4-bit (D0 to D3) of Character Code (Hexadecimal)	0	CG RAM (1)	±		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	1	CG RAM (2)	≡	!	1	A	0	a	n	0	a	0	a	i	"	J	+	y	0	
	2	CG RAM (3)	7	"	2	R	b	n	e	f	6	'	e	s	s	z				
	3	CG RAM (4)	⊂	#	3	C	S	c	s	a	0	0	'	P	g	e	ψ			
	4	CG RAM (5)	⊂	\$	4	D	T	a	t	a	b	a	c	'	e	n	z	o		
	5	CG RAM (6)	⊂	%	5	E	U	e	u	a	b	a	e	s	t	a	n	7		
	6	CG RAM (7)	⊂	&	6	F	V	v	a	0	*	w	0	0	*					
	7	CG RAM (8)	⊂	'	7	G	W	w	S	O	R	X	*	A	L	*				
	8	CG RAM (1)	⊂	(	8	H	X	x	E	0	S	÷	÷	E	K	⊂				
	9	CG RAM (2)	⊂	)	9	I	V	i	w	e	s	i	⊂	⊂	⊂	⊂				
	A	CG RAM (3)	⊂	*		J	Z	z	e	0	a	z	'	Z	P	⊂				
	B	CG RAM (4)	⊂	+		K	k	C	i	R	a	e	L	v	*					
	C	CG RAM (5)	⊂	,		L	\	l	i	a	e	s	⊂	⊂	⊂					
	D	CG RAM (6)	⊂	-		M	m	D	i	a	0	*	'	w	π	⊂				
	E	CG RAM (7)	⊂	.		>	N	n	^	A	0	0	T	0	0	P	⊂			
	F	CG RAM (8)	⊂	/		?	0	_	0	A	A	z	0	'	0	0	0	⊂		



# USING I2C WITH PICAXE

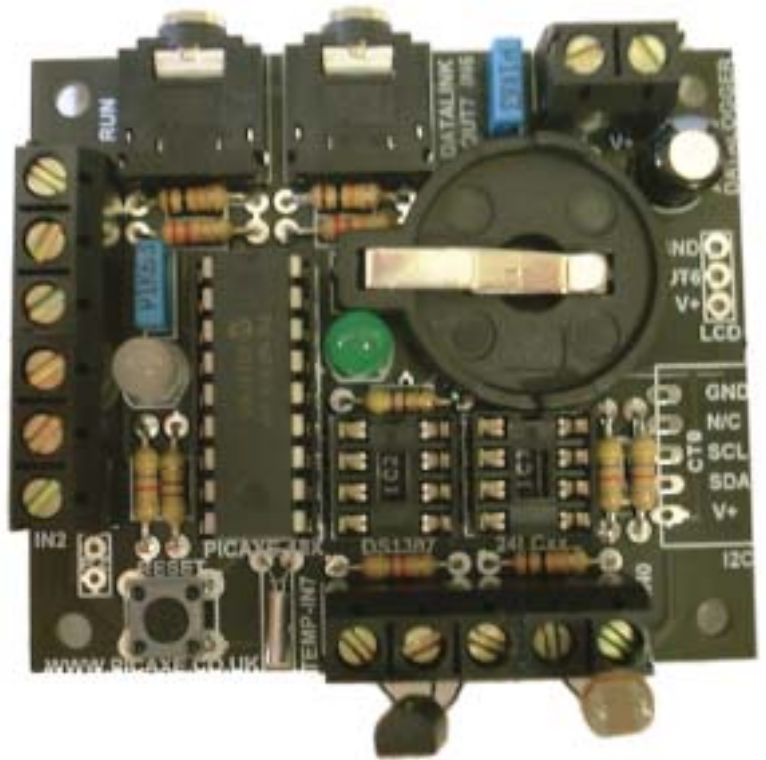
## Contents:

This article provides an introduction into how to use i2c parts with the PICAXE system.

This article:

- 1) Describes the i2c bus
- 2) Explains how the i2c bus is used with the PICAXE system
- 3) Gives an example of using the i2c bus with a 24LCxx series EEPROM
- 4) Gives an example of using the i2c bus with a DS1307 real time clock.
- 5) Gives an example of using the i2c bus with a SPE030 speech synthesizer.

All the information in this datasheet applies to the PICAXE-X parts (18X, 28X, 40X). If you wish to experiment with use of the i2c bus, we recommend use of the AXE110 Datalogger fitted with the AXE034 Real Time Clock Upgrade. This will provide you with a board that has the PICAXE-18X, a 24LC16B EEPROM memory chip, and a DS1307 real-time-clock chip fitted.

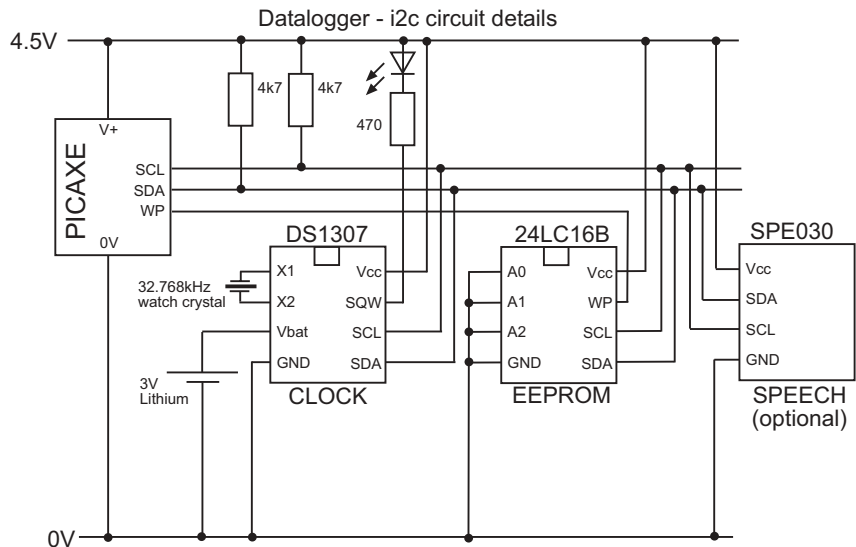


## Terms used in this article:

IC	- integrated circuit or 'chip'
Master	- a microcontroller IC that 'controls' the operation of a circuit
Slave	- a slave IC that does certain specialised tasks for the master IC
Byte	- a number between 0 and 255
Register	- a memory location within the slave that stores 1 byte of data
Register Address	- an address that 'points' to a particular memory register
Block	- group of 256 registers
EEPROM IC	- a slave IC that can store a large amount of data
RTC IC	- a slave IC that can maintain the date / time (real-time-clock)
ADC IC	- a slave IC that can perform analogue-to-digital conversions

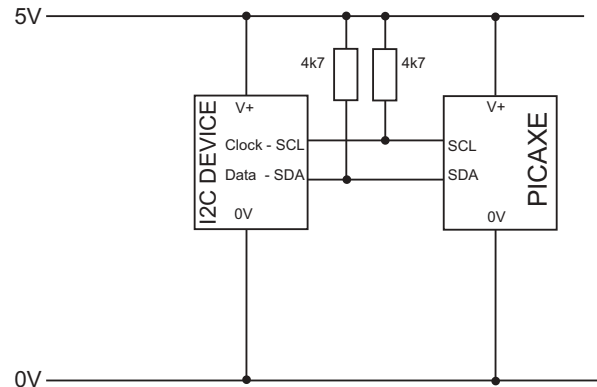
## Sample i2c circuit

PICAXE-18X datalogger circuit.  
Shows connections of  
- 24LC16B EEPROM  
- DS1307 RTC  
- SPE030 Speech Module



## What is the i2c bus?

The Inter-Integrated-Circuit (i2c) bus was originally developed by Phillips Inc. for the transfer of data between ICs at the PCB level. The physical interface of the communication bus consists of just two lines – one for the clock (SCL) and one for the data (SDA). These lines are pulled high by resistors connected to the V+ rail. 4k7 is a commonly used value for these resistors, although the actual value used is not that critical. When either of the master or slave ICs want to ‘transmit’, they pull the lines low by transistors built inside the IC.



The IC that controls the bus is called the Master, and is often a microcontroller – in this article a PICAXE-18X microcontroller will be used as the master device. The other ICs connected to the bus are called Slaves. There can be more than one slave on the bus, as long as each slave has been configured to have a unique ‘slave address’ so that it can be uniquely identified on the bus. In theory there are up to about 112 different addresses available, but most practical applications would generally have between 1 and 10 slave ICs.

## Why use the i2c bus?

### Advantages:

- Most major semiconductor manufacturers produce many low-cost i2c compatible ICs. The range of ICs available is quite extensive - memory EEPROMs, real-time-clocks, ADCs, DACs, PWM motor/fan controllers, LED drivers, digital potentiometers, digital temperature sensors etc. etc.
- Many of these ICs come in small 8 pin packages. This makes the circuit design very straight forward.
- Many slave devices can be connected to the same bus, which only uses two of the microcontroller pins. This is a very efficient use of the microcontroller pins.
- The bus design is very simple, using just two lines and two resistors.

### Disadvantages:

- The i2c bus communication protocol is quite complicated. However this can be easily overcome by using microcontroller systems such as PICAXE, which provide simple BASIC style commands for all the i2c data transfers, and therefore the end user needs no technical knowledge of the bus communication protocols.
- Each slave IC will have a few unique setup parameters (e.g. slave address), which must be extracted from the manufacturers datasheet. This is not normally that difficult, once you know the main parameters that you are looking for!

## Slave Configuration Parameters

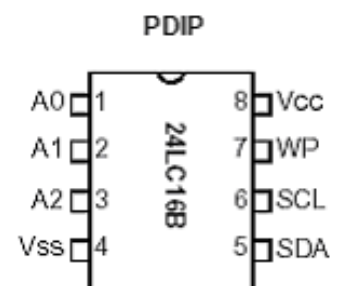
Although all i2c slave devices work in roughly the same way, there are four parameters that must be checked from the manufacturer's datasheet for each slave device used.

### Parameter 1 - Slave Address

As already mentioned, each slave IC on the i2c bus must have a unique address. This is not generally a problem when using different types of IC on the same bus, as most ICs have a different default slave address.

The slave address is generally 7 bits long, with the 8<sup>th</sup> bit reserved to indicate whether the master wishes to write to (1) or read from (0) the slave. A 10-bit slave address is also possible, but is rarely used and so not covered in this article. This means the slave address is often quoted in datasheets as, for instance, 1010000x, with x indicating the read/write bit. When using the PICAXE system the state of this 8<sup>th</sup> bit is not important, as the PICAXE system will automatically set or clear the bit as necessary for a read or a write.

However it is also possible that you may want to use two or more of the same type of IC (e.g. memory EEPROM) on the same bus. This can be overcome by the use of external address pins on the slave device, which can be connected (on the PCB design) to either V+ or 0V to give each slave IC on the PCB a unique address. In the case of the popular 24LCxx series of EEPROMs there are 3 external address pins (A2, A1 and A0). By connecting these pins to V+ or GND on your circuit design, you can ensure that up to 8 parts can be uniquely identified on the same bus.



For these ICs the datasheet slave address may be quoted as, for instance, 1010dddx, where d is 1 or 0 depending on the state of the external address pin A2-A0.

### Parameter 2 -Bus Speed (100 or 400kHz)

The maximum bus speed for data transfer between the master and slave is normally 400kHz. However some parts will only work up to 100kHz, and so the manufacturer datasheet should be checked for each slave IC used. Note this is the maximum speed - all parts can be run at the slower speed if desired.

### Parameter 3 - Register Address Size (Byte or Word)

All data transfer from the master to the slave is a 'write', and this means that a byte of data is transferred from the master to a 'register' within the slave IC. All data transfer from the slave to the master is a 'read'. Simpler slave devices have a maximum of 256 registers, and so a 'register address' of one byte length can be used to identify the particular register of interest. However larger devices, particularly memory EEPROMs, have more than 256 registers and so may need a 'word' (two byte) register address instead.

### Parameter 4 – Page Write Buffer

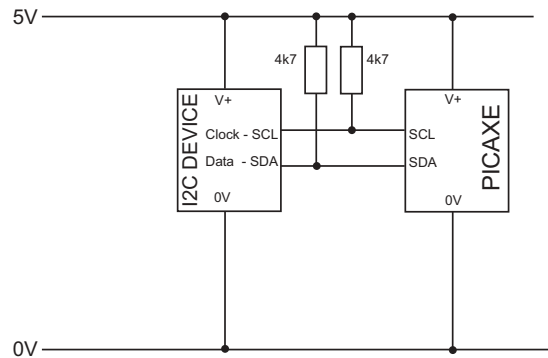
All EEPROM memory chips require a 'write time' to save the data in the chip. This is typically 5 or 10ms. When writing lots of data, this can cause a significant delay. To help overcome this issue, many ICs have a page write buffer that can accept more than one byte at once (typically 8, 16 or 32 bytes) so that all these bytes can be programmed at once. This means, for instance, in the case of 8 bytes you only have one 10ms delay, rather than 80ms of delay. Important Note: One of the biggest mistakes made by beginners is that they don't realise that page writes can only 'start' at a multiple of the buffer size, and cannot overflow the page buffer size. In effect this means (for an 8 byte buffer) you can write 8 bytes up from address 0 (or 8 or 16 etc.) but only up 6 bytes from address 2 (10, 18 etc.), or else you would overflow the 8 byte page write boundary.



## 2) Using i2c with the PICAXE System

### Hardware

All the PICAXE X parts (18X, 28X, 40X) have two pins which can be dedicated for the two i2c communication lines – SDA and SCL. The typical electrical configuration is shown here.



### Software

Communication with the slave device just requires three BASIC commands – `i2cslave`, `readi2c` and `writei2c`

#### `i2cslave`

The `i2cslave` command is used to set up the slave parameters for each slave IC. The syntax is

```
i2cslave slave_address, bus_speed, address_size
```

where `slave_address` is the address (e.g. %10100000)  
`bus_speed` is the keyword `i2cfast` (400kHz) or `i2cslow` (100kHz)  
`address_size` is the keyword `i2cbyte` or `i2cword` as appropriate

#### `writei2c`

The `writei2c` command is used to write data to the slave. The syntax is

```
writei2c start_address, (data,data,data,data...)
```

where `start_address` is the start address (byte or word as appropriate)  
`data` is bytes of data to be sent (either fixed values of variable contents)  
(multiple bytes of data can be sent, but care should be taken not to exceed the page buffer size)

#### `readi2c`

The `readi2c` command is used to read data back from the slave into variables in the PICAXE. The syntax is

```
readi2c start_address, (variable, variable,...)
```

where `start_address` is the start address (byte or word as appropriate)  
`variable` is where the returned data is stored in the master (b0, b1, b2 etc)

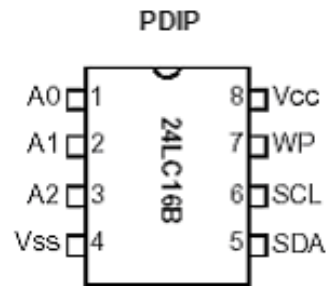
#### Example

To write the text “hello” (actually 5 bytes of data – one byte for each letter) to a 24LC16B memory IC and then read it back into variables, the program would be

```
i2cslave %10100000, i2cfast, i2cbyte    ` set slave parameters
writei2c 0, ("hello")                  ` write the text
pause 10                               ` wait 10ms write time
readi2c 0, (b0,b1,b2,b3,b4)            ` read the data back again
debug b0                               ` display data on screen
```

### 3) Practical Example – 24LC16B EEPROM

Many projects involve the storage of data. This may be data collected during a datalogging experiment, or pre-configured data built into the circuit at the time of build (e.g. messages in different languages to be displayed on an LCD screen). The PICAXE chips can generally store 128 or 256 bytes of data internally, but some projects may require much more than this, and so an external memory storage IC is required.



External EEPROM (electrically-erasable-programmable-read-only-memories) ICs can be used to store large amounts of data. Most EEPROMs store data in 'blocks' of 256 registers, each register storing one byte of data. Simplest EEPROMs may only have one block of 256 registers, more expensive EEPROMs can have up to 256 blocks, giving a total of  $256 \times 256 = 65536$  (64k) memory registers.

The 24LCxx series EEPROMs are probably the most commonly used i2c EEPROM devices. Many manufacturers make these parts, but we will only consider Microchip brand parts in this article because these tend to be readily available via mail order catalogues. These EEPROMs can be written to over 1 million times, and the EEPROM memory also retains data when the power is removed. Pin 7 of the IC is a write-enable pin that can prevent the data being corrupted (keep the pin high to prevent data being changed). Often this pin is connected to a microcontroller pin, so that the microcontroller can control when data can be written (pull pin low to enable writes).

The cheapest EEPROMs (e.g. Microchip parts ending in the letter 'B') only use a single byte register address, which by definition can only uniquely identify 256 registers. This means that the various blocks (if they exist) must be identified in a different way. The 24LC16B has 8 blocks, the other EEPROMs have less (see table below). The way these cheap EEPROMs overcome this address problem is by merging the block address into the slave address. This means, in effect, that a single 24LC16B 'appears' on the i2c bus as 8 different 'slaves', each slave having a unique address and containing 256 registers. This system might at first appear quite strange, but the IC is constructed this way to keep the cost of the IC to a minimum. However this system does have the downfall that only one part can be used per bus (the external IC pins A2-A0 are not actually physically connected within these cheaper 'B' parts).

The more expensive EEPROMs (24LC32 upwards) use a word register address, and so the block address can be incorporated within the normal register word address. This means the EEPROM appears on the i2c bus as a single slave, and so up to 8 identical devices can be connected to the bus by configuring the external A2 to A0 address pins accordingly. Using 8 of the commonly available 24LC256 EEPROMs will give a huge 2Mb of memory!

Device	Registers	Buffer	Slave	Speed	Address
24LC01B	128	8	%1010xxxx	i2cfast (400kHz)	i2cbyte
24LC02B	256	8	%1010xxxx	i2cfast (400kHz)	i2cbyte
24LC04B	512	16	%1010xxbx	i2cfast (400kHz)	i2cbyte
24LC08B	1k (1024)	16	%1010xbbx	i2cfast (400kHz)	i2cbyte
24LC16B	2k (2048)	16	%1010bbbx	i2cfast (400kHz)	i2cbyte
24LC32A	4k (4096)	32	%1010dddx	i2cfast (400kHz)	i2cword
24LC65	8k (8192)	64	%1010dddx	i2cfast (400kHz)	i2cword
24LC128	16k (16384)	64	%1010dddx	i2cfast (400kHz)	i2cword
24LC256	32k (32768)	64	%1010dddx	i2cfast (400kHz)	i2cword
24LC512	64k (65536)	128	%1010dddx	i2cfast (400kHz)	i2cword

where b = block address (internal to EEPROM)  
 d = device address (configured by external pins A2, A1, A0)  
 x = don't care

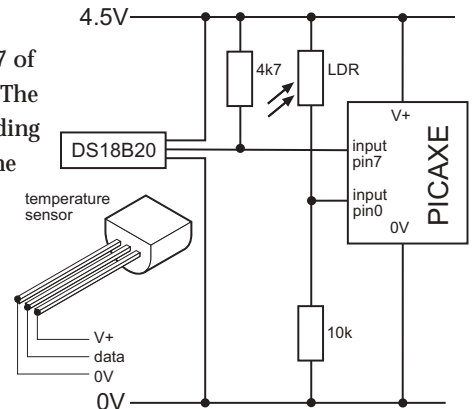
## Saving Data

The following example reads the data from a digital temperature sensor (on input 7 of the PICAXE master) and a LDR light sensor (on input 0) every minute for an hour. The data is saved in a 24LC04B, 24LC08B or 24LC16B EEPROM. Each temperature reading is saved in the first block (000) of the memory, and each light reading is saved in the second block (001) of the memory. A for..next loop is used to repeat the action 60 times, and the loop counter value (0 to 59) is used as the address to save the data within the appropriate memory block.

```

for b1 = 0 to 59                ` start for..next loop
  readtemp 7,b2                 ` read temp value from 7
  i2cslave %10100000, i2cfast, i2cbyte ` set block0 parameters
  writei2c b1,(b2)              ` write the value
  pause 10                      ` wait EEPROM write time
  readadc 0,b3                  ` read light value from 0
  i2cslave %10100010, i2cfast, i2cbyte ` set block1 parameters
  writei2c b1,(b3)              ` write the value
  pause 60000                   ` wait 1 minute
next b1                          ` next loop

```



## Reading Data

The following example reads back the data saved in the example above, and displays the data on a Serial LCD Module (part AXE033). The serout command is the command that transmits the data from the PICAXE master to the serial LCD module (connected on output 6). The temperature value is shown on the top line of the display, the light value on the bottom line of the display. Each reading is displayed for 2 seconds.

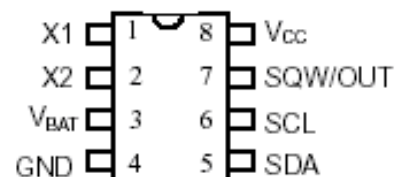
```

for b1 = 0 to 59                ` start for..next loop
  i2cslave %10100000, i2cfast, i2cbyte ` set block0 parameters
  readi2c b1,(b2)                ` read the temp value
  i2cslave %10100010, i2cfast, i2cbyte ` set block1 parameters
  readi2c b1,(b3)                ` read the light value
  serout 6,N2400,(254,128,"Temp Value =",#b2, " ") ` display temp
  serout 6,N2400,(254,192,"Light Value =",#b3, " ") ` display light
  pause 2000                      ` wait 2 seconds
next b1                          ` next loop

```

#### 4) Practical Example – DS1307 Real Time Clock

The Maxim/Dallas Semiconductor DS1307 is an accurate real-time-clock, which automatically maintains the current time and date, including compensation for months with less than 31 days and leap years. The DS1307 is an 8 pin device, and a standard low-cost 32.768 kHz 12pF quartz watch crystal is connected to pins 1 and 2 to provide the accurate time base. An optional 3V lithium 'backup' cell can also be connected to pin 3, this ensures that the time is kept up to date when the main circuit power is removed. The IC automatically detects removal of the main power source and moves to the lithium cell power as and when required. The cell should last at least ten years.



DS1307 8-Pin DIP (300 mil)

The DS1307 also has two additional features of interest. Pin 7 is an open collector output that can be programmed to 'flash' at 1Hz. This allows an LED to be attached as a 'seconds indicator' in clock applications. The IC also contains 56 bytes of general purpose RAM, which can be used as extra memory by the master if required.

From the manufacturers datasheet for the DS1307 ([www.dalsemi.com](http://www.dalsemi.com)), the following i2c details can be found:

slave address       - 1101000x  
address size        - 1 byte  
bus speed           - 100kHz

The registers of the DS1307 are defined as follows:

Address	Register
00	Seconds (0-59)
01	Minutes (0-59)
02	Hours (0-23)
03	Day of Week (1-7)
04	Date (1-31)
05	Month (1-12)
06	Year (00-99)
07	Control (set to 16 (\$10))
08-\$3F	General Purpose RAM

All the time/date data is in BCD (binary-coded-decimal) format, which makes it very easy to read and write using hex notation e.g. 11:35am will contain \$11 in the hours register and \$35 in the minutes register.

## Setting the Time / Date

To set the correct time after the circuit is first powered up, the current time must be written to the registers. The following example PICAXE program will setup the time to 11:59:00 on Thursday 25/12/03.

This is carried out by loading the registers in order from address 00 upwards i.e. seconds then minutes then hours etc.

```
i2cslave %11010000, i2cslow, i2cbyte
writei2c 0, ($00, $59, $11, $03, $25, $12, $03, $10)
end
```

## Reading the Time / Date

To read the current time you can use the following program to load variables within the PICAXE with the various register values from the DS1307. Calculations can then be carried out to see, for instance, if a particular alarm point has been reached. This example program acts as an alarm clock, checking the time every 30 seconds. If the time is exactly 07:00 then a buzzer, connected to output 7, will sound for 20 seconds.

```
i2cslave %11010000, i2cslow, i2cbyte      ` set slave parameter

loop:
  pause 30000                             ` wait 30 sec
  readi2c 0, (b0, b1, b2)                 ` read sec, min, hour
  if b2 <> $07 then loop                   ` if hour not 7 loop
  if b1 <> $00 then loop                   ` if min not 00 loop

  high 7                                   ` switch on buzzer
  pause 20000                              ` wait 20 sec
  low 7                                    ` switch off buzzer
  pause 60000                              ` wait 60 sec to prevent repeat
  goto loop                                ` loop
```

## 5) Practical Example – SPE030 Speech module.

The SPE030 module is a speech synthesizer that will speak the text sent to it over the i2c bus.

From the SPE030 datasheet (spe030.pdf), the following i2c details can be found:

```
slave address    - $C4
address size     - 1 byte
bus speed        - 400kHz
```

The following program will generate the speech "hello PICAXE user". For further information see the SPE030 datasheet (spe030.pdf).

```
i2cslave $C4,i2cfast,i2cbyte
writei2c 0,(0,0,5,3,"hello pickacks user",0)
writei2c 0,(64)
```

Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	<u>SP</u>
010 0001	041	33	21	<u>!</u>
010 0010	042	34	22	<u>"</u>
010 0011	043	35	23	<u>#</u>
010 0100	044	36	24	<u>\$</u>
010 0101	045	37	25	<u>%</u>
010 0110	046	38	26	<u>&amp;</u>
010 0111	047	39	27	<u>'</u>
010 1000	050	40	28	<u>(</u>
010 1001	051	41	29	<u>)</u>
010 1010	052	42	2A	<u>*</u>
010 1011	053	43	2B	<u>±</u>
010 1100	054	44	2C	<u>ˆ</u>
010 1101	055	45	2D	<u>˜</u>
010 1110	056	46	2E	<u>.</u>
010 1111	057	47	2F	<u>/</u>
011 0000	060	48	30	<u>0</u>
011 0001	061	49	31	<u>1</u>
011 0010	062	50	32	<u>2</u>

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	<u>@</u>
100 0001	101	65	41	<u>A</u>
100 0010	102	66	42	<u>B</u>
100 0011	103	67	43	<u>C</u>
100 0100	104	68	44	<u>D</u>
100 0101	105	69	45	<u>E</u>
100 0110	106	70	46	<u>F</u>
100 0111	107	71	47	<u>G</u>
100 1000	110	72	48	<u>H</u>
100 1001	111	73	49	<u>I</u>
100 1010	112	74	4A	<u>J</u>
100 1011	113	75	4B	<u>K</u>
100 1100	114	76	4C	<u>L</u>
100 1101	115	77	4D	<u>M</u>
100 1110	116	78	4E	<u>N</u>
100 1111	117	79	4F	<u>O</u>
101 0000	120	80	50	<u>P</u>
101 0001	121	81	51	<u>Q</u>
101 0010	122	82	52	<u>R</u>

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	<u>`</u>
110 0001	141	97	61	<u>a</u>
110 0010	142	98	62	<u>b</u>
110 0011	143	99	63	<u>c</u>
110 0100	144	100	64	<u>d</u>
110 0101	145	101	65	<u>e</u>
110 0110	146	102	66	<u>f</u>
110 0111	147	103	67	<u>g</u>
110 1000	150	104	68	<u>h</u>
110 1001	151	105	69	<u>i</u>
110 1010	152	106	6A	<u>j</u>
110 1011	153	107	6B	<u>k</u>
110 1100	154	108	6C	<u>l</u>
110 1101	155	109	6D	<u>m</u>
110 1110	156	110	6E	<u>n</u>
110 1111	157	111	6F	<u>o</u>
111 0000	160	112	70	<u>p</u>
111 0001	161	113	71	<u>q</u>
111 0010	162	114	72	<u>r</u>

011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	:
011 1100	074	60	3C	≤
011 1101	075	61	3D	≡
011 1110	076	62	3E	≥
011 1111	077	63	3F	?

101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	↓
111 1101	175	125	7D	↑
111 1110	176	126	7E	~




[Home](#)
[Protocol](#)
[profile](#) | [re](#)

Forums | [No new posts please! 4](#) |  
 DS1307+Correction

[Post Reply](#)
[Send Top](#)
[Show topics from last day](#)
**Author**
**Topic**

tarzan

 Posted - 16 March 2004 16:49   

After some head scratching here is the corrected file for DS1307.bas  
 It's the tens digit that needed attention ( / 16 ).  
 Technical please note.

; Example of how to use DS1307 Time Clock (i2c device)  
 ; Note the data is sent/received in BCD format.

```
symbol seconds = b0
symbol mins = b1
symbol hour = b2
symbol day = b3
symbol date = b4
symbol month = b5
symbol year = b6
symbol control = b7
symbol temp = b8
```

```
' set DS1307 slave address
i2cslave %11010000, i2cslow, i2cbyte
```

```
' uncomment this line to update the clock time
' goto start_clock
```

```
' read time and date and display on serial LCD module
```

```
main:
readi2c 0,(seconds,mins,hour,day,date,month,year)
```

```
' debug b0 '(optional debug computer to screen)
```

```
serout 7,N2400,(254,128) 'start of first line
```

```
let temp = date & %00110000 / 16
serout 7,N2400,(#temp)
let temp = date & %00001111
serout 7,N2400,(#temp,"/")
```

```
let temp = month & %0001000 / 16
serout 7,N2400,(#temp)
let temp = month & %00001111
serout 7,N2400,(#temp,"/")
```

```
let temp = year & %11110000 / 16
```

```

serout 7,N2400,(#temp)
let temp = year & %00001111
serout 7,N2400,(#temp," ")

let temp = hour & %00110000 / 16
serout 7,N2400,(#temp)
let temp = hour & %00001111
serout 7,N2400,(#temp,":")

let temp = mins & %01110000 / 16
serout 7,N2400,(#temp)
let temp = mins & %00001111
serout 7,N2400,(#temp,":")

let temp = seconds & %01110000 / 16
serout 7,N2400,(#temp)
let temp = seconds & %00001111
serout 7,N2400,(#temp)

pause 5000
goto main

'write time and date e.g. to 11:59:00 on Thurs 25/12/03
start_clock:
let seconds = $00 ' 00 Note all BCD format
let mins = $59 ' 59 Note all BCD format
let hour = $11 ' 11 Note all BCD format
let day = $03 ' 03 Note all BCD format
let date = $25 ' 25 Note all BCD format
let month = $12 ' 12 Note all BCD format
let year = $03 ' 03 Note all BCD format
let control = %00010000 ' Enable output at 1Hz

writei2c 0,(seconds,mins,hour,day,date,month,year,control)

end

I prefer using two lines so you can see the seconds.

; Example of how to use DS1307 Time Clock (i2c device)
; Note the data is sent/received in BCD format.

symbol seconds = b0
symbol mins = b1
symbol hour = b2
symbol day = b3
symbol date = b4
symbol month = b5
symbol year = b6
symbol control = b7
symbol temp = b8

' set DS1307 slave address
i2cslave %11010000, i2cslow, i2cbyte

' uncomment this line to update the clock time

```

```
' goto start_clock

' read time and date and display on serial LCD module
init:
serout 7,N2400,(254,1) 'clear LCD
pause 30
main:

readi2c 0,(seconds,mins,hour,day,date,month,year)

'debug b0 '(optional debug computer to screen

serout 7,N2400,(254,192)

let temp = date & %00110000 / 16
serout 7,N2400,(#temp)
let temp = date & %00001111
serout 7,N2400,(#temp,"/")

let temp = month & %00001000 / 16
serout 7,N2400,(#temp)
let temp = month & %00001111
serout 7,N2400,(#temp,"/")

let temp = year & %11110000 / 16
serout 7,N2400,(#temp)
let temp = year & %00001111
serout 7,N2400,(#temp," ")

serout 7,N2400,(254,128)

let temp = hour & %00110000 / 16
serout 7,N2400,(#temp)
let temp = hour & %00001111
serout 7,N2400,(#temp,":")

let temp = mins & %01110000 / 16
serout 7,N2400,(#temp)
let temp = mins & %00001111
serout 7,N2400,(#temp,":")

let temp = seconds & %01110000 / 16
serout 7,N2400,(#temp)
let temp = seconds & %00001111
serout 7,N2400,(#temp)

pause 100
goto main

'write time and date e.g. to 11:59:00 on Thurs 25/12/03
start_clock:
let seconds = $00 ' 00 Note all BCD format
let mins = $59 ' 59 Note all BCD format
let hour = $11 ' 11 Note all BCD format
let day = $03 ' 03 Note all BCD format
```

```
let date = $25 ' 25 Note all BCD format
let month = $12 ' 12 Note all BCD format
let year = $03 ' 03 Note all BCD format
let control = %00010000 ' Enable output at 1Hz
```

```
writei2c 0,(seconds,mins,hour,day,date,month,year,control)
goto main
```

```
end
```

Edited by - tarzan on 3/16/2004 5:15:48 PM

Edited by - tarzan on 3/16/2004 5:57:39 PM

Click [Here](#) To Close Thread, Administrators & Mod

[Show All Forums](#) | [Post Reply](#)

Revolution Education Ltd, 4 Old Dairy Business Centre, Melcombe Road, Bath, BA2 3LR  
Tel: +44 (0)1225 340563 Fax: +44 (0)1225 340564 Email: [info@rev-ed.co.uk](mailto:info@rev-ed.co.uk)

### FEATURES

- Real-time clock (RTC) counts seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation valid up to 2100
- 56-byte, battery-backed, nonvolatile (NV) RAM for data storage
- Two-wire serial interface
- Programmable squarewave output signal
- Automatic power-fail detect and switch circuitry
- Consumes less than 500nA in battery backup mode with oscillator running
- Optional industrial temperature range: -40°C to +85°C
- Available in 8-pin DIP or SOIC
- Underwriters Laboratory (UL) recognized

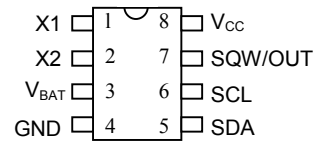
### ORDERING INFORMATION

DS1307	8-Pin DIP (300-mil)
DS1307Z	8-Pin SOIC (150-mil)
DS1307N	8-Pin DIP (Industrial)
DS1307ZN	8-Pin SOIC (Industrial)

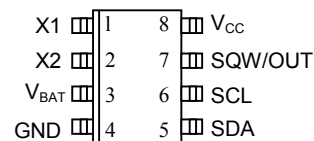
### DESCRIPTION

The DS1307 Serial Real-Time Clock is a low-power, full binary-coded decimal (BCD) clock/calendar plus 56 bytes of NV SRAM. Address and data are transferred serially via a 2-wire, bi-directional bus. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with AM/PM indicator. The DS1307 has a built-in power sense circuit that detects power failures and automatically switches to the battery supply.

### PIN ASSIGNMENT



DS1307 8-Pin DIP (300-mil)

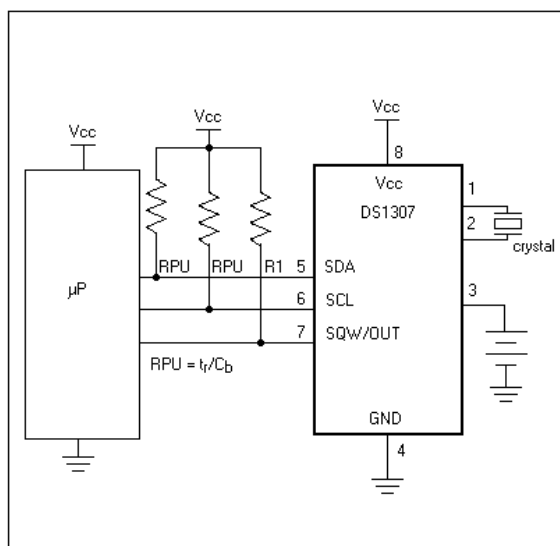


DS1307 8-Pin SOIC (150-mil)

### PIN DESCRIPTION

V <sub>CC</sub>	- Primary Power Supply
X1, X2	- 32.768kHz Crystal Connection
V <sub>BAT</sub>	- +3V Battery Input
GND	- Ground
SDA	- Serial Data
SCL	- Serial Clock
SQW/OUT	- Square Wave/Output Driver

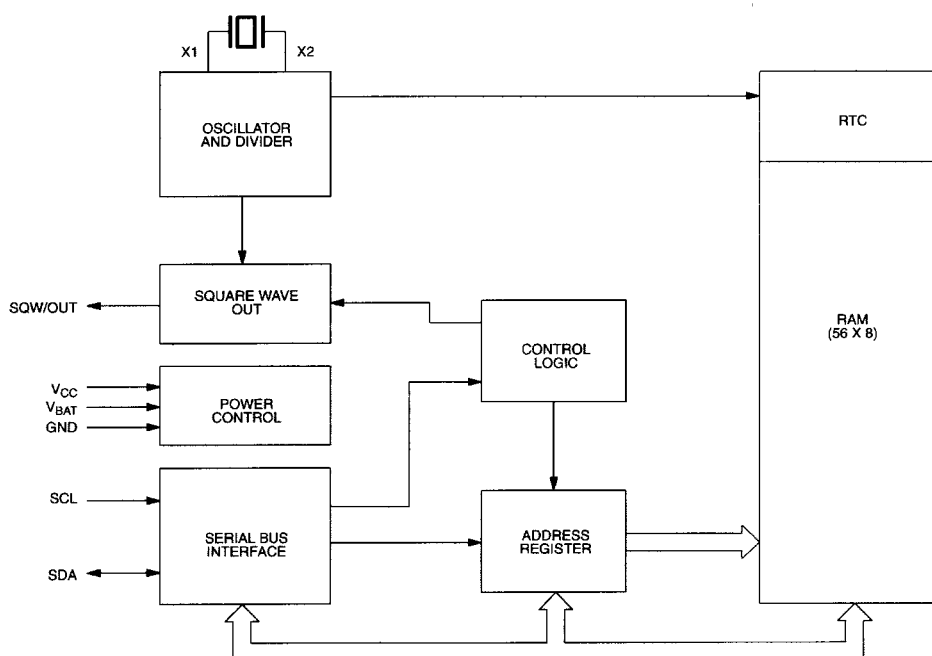
## TYPICAL OPERATING CIRCUIT



## OPERATION

The DS1307 operates as a slave device on the serial bus. Access is obtained by implementing a START condition and providing a device identification code followed by a register address. Subsequent registers can be accessed sequentially until a STOP condition is executed. When  $V_{CC}$  falls below  $1.25 \times V_{BAT}$  the device terminates an access in progress and resets the device address counter. Inputs to the device will not be recognized at this time to prevent erroneous data from being written to the device from an out of tolerance system. When  $V_{CC}$  falls below  $V_{BAT}$  the device switches into a low-current battery backup mode. Upon power-up, the device switches from battery to  $V_{CC}$  when  $V_{CC}$  is greater than  $V_{BAT} + 0.2V$  and recognizes inputs when  $V_{CC}$  is greater than  $1.25 \times V_{BAT}$ . The block diagram in Figure 1 shows the main elements of the serial RTC.

## DS1307 BLOCK DIAGRAM Figure 1



## SIGNAL DESCRIPTIONS

**V<sub>CC</sub>, GND** – DC power is provided to the device on these pins. V<sub>CC</sub> is the +5V input. When 5V is applied within normal limits, the device is fully accessible and data can be written and read. When a 3V battery is connected to the device and V<sub>CC</sub> is below 1.25 x V<sub>BAT</sub>, reads and writes are inhibited. However, the timekeeping function continues unaffected by the lower input voltage. As V<sub>CC</sub> falls below V<sub>BAT</sub> the RAM and timekeeper are switched over to the external power supply (nominal 3.0V DC) at V<sub>BAT</sub>.

**V<sub>BAT</sub>** – Battery input for any standard 3V lithium cell or other energy source. Battery voltage must be held between 2.0V and 3.5V for proper operation. The nominal write protect trip point voltage at which access to the RTC and user RAM is denied is set by the internal circuitry as 1.25 x V<sub>BAT</sub> nominal. A lithium battery with 48mAh or greater will back up the DS1307 for more than 10 years in the absence of power at 25°C. UL recognized to ensure against reverse charging current when used in conjunction with a lithium battery.

See “Conditions of Acceptability” at <http://www.maxim-ic.com/TechSupport/QA/ntrl.htm>.

**SCL (Serial Clock Input)** – SCL is used to synchronize data movement on the serial interface.

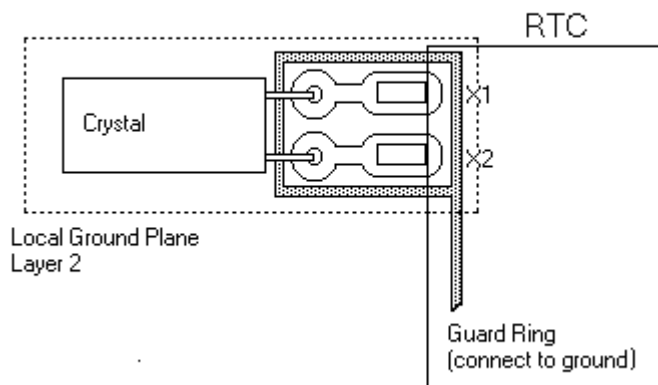
**SDA (Serial Data Input/Output)** – SDA is the input/output pin for the 2-wire serial interface. The SDA pin is open drain which requires an external pullup resistor.

**SQW/OUT (Square Wave/Output Driver)** – When enabled, the SQWE bit set to 1, the SQW/OUT pin outputs one of four square wave frequencies (1Hz, 4kHz, 8kHz, 32kHz). The SQW/OUT pin is open drain and requires an external pull-up resistor. SQW/OUT will operate with either V<sub>cc</sub> or V<sub>bat</sub> applied.

**X1, X2** – Connections for a standard 32.768kHz quartz crystal. The internal oscillator circuitry is designed for operation with a crystal having a specified load capacitance (CL) of 12.5pF.

For more information on crystal selection and crystal layout considerations, please consult Application Note 58, “Crystal Considerations with Dallas Real-Time Clocks.” The DS1307 can also be driven by an external 32.768kHz oscillator. In this configuration, the X1 pin is connected to the external oscillator signal and the X2 pin is floated.

## RECOMMENDED LAYOUT FOR CRYSTAL





## CLOCK ACCURACY

The accuracy of the clock is dependent upon the accuracy of the crystal and the accuracy of the match between the capacitive load of the oscillator circuit and the capacitive load for which the crystal was trimmed. Additional error will be added by crystal frequency drift caused by temperature shifts. External circuit noise coupled into the oscillator circuit may result in the clock running fast. See Application Note 58, “Crystal Considerations with Dallas Real-Time Clocks” for detailed information.

Please review Application Note 95, “Interfacing the DS1307 with a 8051-Compatible Microcontroller” for additional information.

## RTC AND RAM ADDRESS MAP

The address map for the RTC and RAM registers of the DS1307 is shown in Figure 2. The RTC registers are located in address locations 00h to 07h. The RAM registers are located in address locations 08h to 3Fh. During a multi-byte access, when the address pointer reaches 3Fh, the end of RAM space, it wraps around to location 00h, the beginning of the clock space.

### DS1307 ADDRESS MAP Figure 2

00H	SECONDS
	MINUTES
	HOURS
	DAY
	DATE
	MONTH
	YEAR
07H	CONTROL
08H	RAM
3FH	56 x 8

## CLOCK AND CALENDAR

The time and calendar information is obtained by reading the appropriate register bytes. The RTC registers are illustrated in Figure 3. The time and calendar are set or initialized by writing the appropriate register bytes. The contents of the time and calendar registers are in the BCD format. Bit 7 of register 0 is the clock halt (CH) bit. When this bit is set to a 1, the oscillator is disabled. When cleared to a 0, the oscillator is enabled.

**Please note that the initial power-on state of all registers is not defined. Therefore, it is important to enable the oscillator (CH bit = 0) during initial configuration.**

The DS1307 can be run in either 12-hour or 24-hour mode. Bit 6 of the hours register is defined as the 12- or 24-hour mode select bit. When high, the 12-hour mode is selected. In the 12-hour mode, bit 5 is the AM/PM bit with logic high being PM. In the 24-hour mode, bit 5 is the second 10 hour bit (20-23 hours).

On a 2-wire START, the current time is transferred to a second set of registers. The time information is read from these secondary registers, while the clock may continue to run. This eliminates the need to re-read the registers in case of an update of the main registers during a read.

## DS1307 TIMEKEEPER REGISTERS Figure 3

BIT7										BIT0			
00H	CH	10 SECONDS				SECONDS						00-59	
	0	10 MINUTES				MINUTES						00-59	
	0	12 24	10 HR A/P	10 HR		HOURS						01-12 00-23	
	0	0	0	0	0	DAY						1-7	
	0	0	10 DATE		DATE							01-28/29 01-30 01-31	
	0	0	0	10 MONTH	MONTH							01-12	
	10 YEAR				YEAR								00-99
07H	OUT	0	0	SQWE	0	0	RS1	RS0					

### CONTROL REGISTER

The DS1307 control register is used to control the operation of the SQW/OUT pin.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
OUT	0	0	SQWE	0	0	RS1	RS0

**OUT (Output control):** This bit controls the output level of the SQW/OUT pin when the square wave output is disabled. If SQWE = 0, the logic level on the SQW/OUT pin is 1 if OUT = 1 and is 0 if OUT = 0.

**SQWE (Square Wave Enable):** This bit, when set to a logic 1, will enable the oscillator output. The frequency of the square wave output depends upon the value of the RS0 and RS1 bits. With the square wave output set to 1Hz, the clock registers update on the falling edge of the square wave.

**RS (Rate Select):** These bits control the frequency of the square wave output when the square wave output has been enabled. Table 1 lists the square wave frequencies that can be selected with the RS bits.

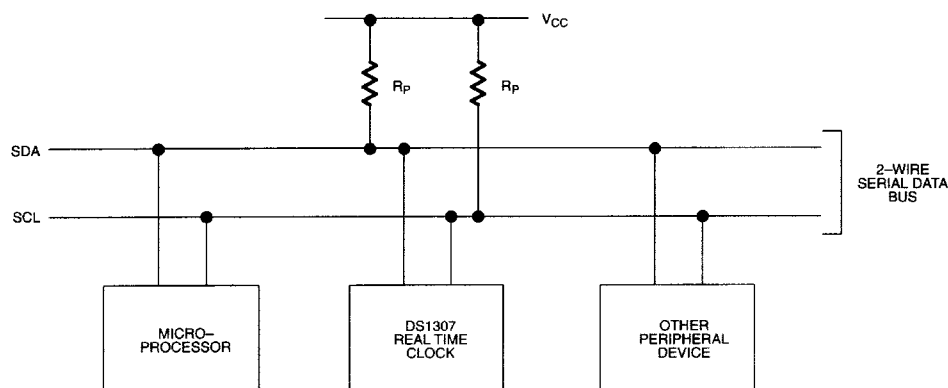
### SQUAREWAVE OUTPUT FREQUENCY Table 1

RS1	RS0	SQW OUTPUT FREQUENCY
0	0	1Hz
0	1	4.096kHz
1	0	8.192kHz
1	1	32.768kHz

## 2-WIRE SERIAL DATA BUS

The DS1307 supports a bi-directional, 2-wire bus and data transmission protocol. A device that sends data onto the bus is defined as a transmitter and a device receiving data as a receiver. The device that controls the message is called a master. The devices that are controlled by the master are referred to as slaves. The bus must be controlled by a master device that generates the serial clock (SCL), controls the bus access, and generates the START and STOP conditions. The DS1307 operates as a slave on the 2-wire bus. A typical bus configuration using this 2-wire protocol is shown in Figure 4.

### TYPICAL 2-WIRE BUS CONFIGURATION Figure 4



Figures 5, 6, and 7 detail how data is transferred on the 2-wire bus.

- Data transfer may be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is HIGH. Changes in the data line while the clock line is high will be interpreted as control signals.

Accordingly, the following bus conditions have been defined:

**Bus not busy:** Both data and clock lines remain HIGH.

**Start data transfer:** A change in the state of the data line, from HIGH to LOW, while the clock is HIGH, defines a START condition.

**Stop data transfer:** A change in the state of the data line, from LOW to HIGH, while the clock line is HIGH, defines the STOP condition.

**Data valid:** The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.

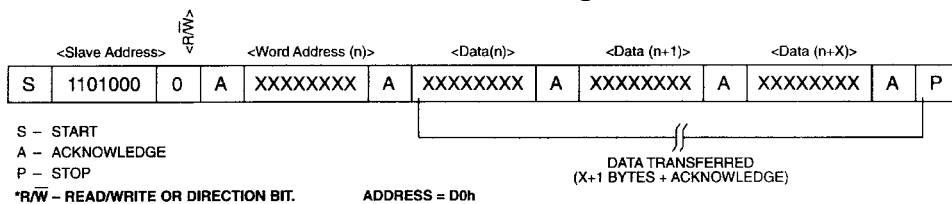
Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between START and STOP conditions is not limited, and is determined by the master device. The information is transferred byte-wise and each receiver acknowledges with a ninth bit. Within the 2-wire bus specifications a regular mode (100kHz clock rate) and a fast mode (400kHz clock rate) are defined. The DS1307 operates in the regular mode (100kHz) only.



The DS1307 may operate in the following two modes:

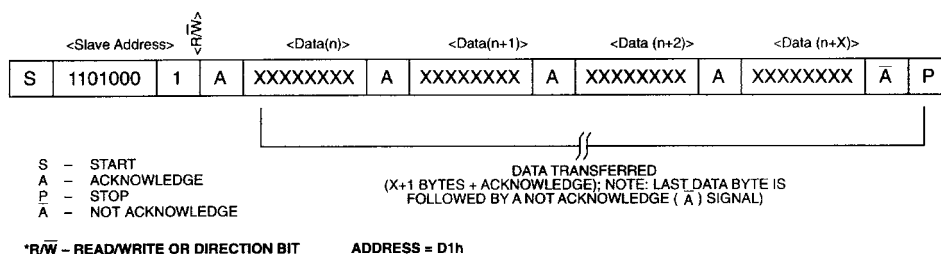
1. **Slave receiver mode (DS1307 write mode):** Serial data and clock are received through SDA and SCL. After each byte is received an acknowledge bit is transmitted. START and STOP conditions are recognized as the beginning and end of a serial transfer. Address recognition is performed by hardware after reception of the slave address and \*direction bit (See Figure 6). The address byte is the first byte received after the start condition is generated by the master. The address byte contains the 7 bit DS1307 address, which is 1101000, followed by the \*direction bit ( $R/\overline{W}$ ) which, for a write, is a 0. After receiving and decoding the address byte the device outputs an acknowledge on the SDA line. After the DS1307 acknowledges the slave address + write bit, the master transmits a register address to the DS1307. This will set the register pointer on the DS1307. The master will then begin transmitting each byte of data with the DS1307 acknowledging each byte received. The master will generate a stop condition to terminate the data write.

## DATA WRITE – SLAVE RECEIVER MODE Figure 6



2. **Slave transmitter mode (DS1307 read mode):** The first byte is received and handled as in the slave receiver mode. However, in this mode, the \*direction bit will indicate that the transfer direction is reversed. Serial data is transmitted on SDA by the DS1307 while the serial clock is input on SCL. START and STOP conditions are recognized as the beginning and end of a serial transfer (See Figure 7). The address byte is the first byte received after the start condition is generated by the master. The address byte contains the 7-bit DS1307 address, which is 1101000, followed by the \*direction bit ( $R/\overline{W}$ ) which, for a read, is a 1. After receiving and decoding the address byte the device inputs an acknowledge on the SDA line. The DS1307 then begins to transmit data starting with the register address pointed to by the register pointer. If the register pointer is not written to before the initiation of a read mode the first address that is read is the last one stored in the register pointer. The DS1307 must receive a “not acknowledge” to end a read.

## DATA READ – SLAVE TRANSMITTER MODE Figure 7



**ABSOLUTE MAXIMUM RATINGS\***

Voltage on Any Pin Relative to Ground	-0.5V to +7.0V
Storage Temperature	-55°C to +125°C
Soldering Temperature	260°C for 10 seconds DIP See JPC/JEDEC Standard J-STD-020A for Surface Mount Devices

\* This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operation sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect reliability.

Range	Temperature	V <sub>CC</sub>
Commercial	0°C to +70°C	4.5V to 5.5V V <sub>CC1</sub>
Industrial	-40°C to +85°C	4.5V to 5.5V V <sub>CC1</sub>

**RECOMMENDED DC OPERATING CONDITIONS**

(Over the operating range\*)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Supply Voltage	V <sub>CC</sub>	4.5	5.0	5.5	V	
Logic 1	V <sub>IH</sub>	2.2		V <sub>CC</sub> + 0.3	V	
Logic 0	V <sub>IL</sub>	-0.5		+0.8	V	
V <sub>BAT</sub> Battery Voltage	V <sub>BAT</sub>	2.0		3.5	V	

\*Unless otherwise specified.

**DC ELECTRICAL CHARACTERISTICS**

(Over the operating range\*)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Input Leakage (SCL)	I <sub>LI</sub>			1	μA	
I/O Leakage (SDA & SQW/OUT)	I <sub>LO</sub>			1	μA	
Logic 0 Output (I <sub>OL</sub> = 5mA)	V <sub>OL</sub>			0.4	V	
Active Supply Current	I <sub>CCA</sub>			1.5	mA	7
Standby Current	I <sub>CCS</sub>			200	μA	1
Battery Current (OSC ON); SQW/OUT OFF	I <sub>BAT1</sub>		300	500	nA	2
Battery Current (OSC ON); SQW/OUT ON (32kHz)	I <sub>BAT2</sub>		480	800	nA	
Power-Fail Voltage	V <sub>PF</sub>	1.216 x V <sub>BAT</sub>	1.25 x V <sub>BAT</sub>	1.284 x V <sub>BAT</sub>	V	8

\*Unless otherwise specified.

**AC ELECTRICAL CHARACTERISTICS**

(Over the operating range\*)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
SCL Clock Frequency	$f_{SCL}$	0		100	kHz	
Bus Free Time Between a STOP and START Condition	$t_{BUF}$	4.7			$\mu$ s	
Hold Time (Repeated) START Condition	$t_{HD:STA}$	4.0			$\mu$ s	3
LOW Period of SCL Clock	$t_{LOW}$	4.7			$\mu$ s	
HIGH Period of SCL Clock	$t_{HIGH}$	4.0			$\mu$ s	
Set-up Time for a Repeated START Condition	$t_{SU:STA}$	4.7			$\mu$ s	
Data Hold Time	$t_{HD:DAT}$	0			$\mu$ s	4,5
Data Set-up Time	$t_{SU:DAT}$	250			ns	
Rise Time of Both SDA and SCL Signals	$t_R$			1000	ns	
Fall Time of Both SDA and SCL Signals	$t_F$			300	ns	
Set-up Time for STOP Condition	$t_{SU:STO}$	4.7			$\mu$ s	
Capacitive Load for each Bus Line	$C_B$			400	pF	6
I/O Capacitance ( $T_A = 25^\circ\text{C}$ )	$C_{I/O}$		10		pF	
Crystal Specified Load Capacitance ( $T_A = 25^\circ\text{C}$ )			12.5		pF	

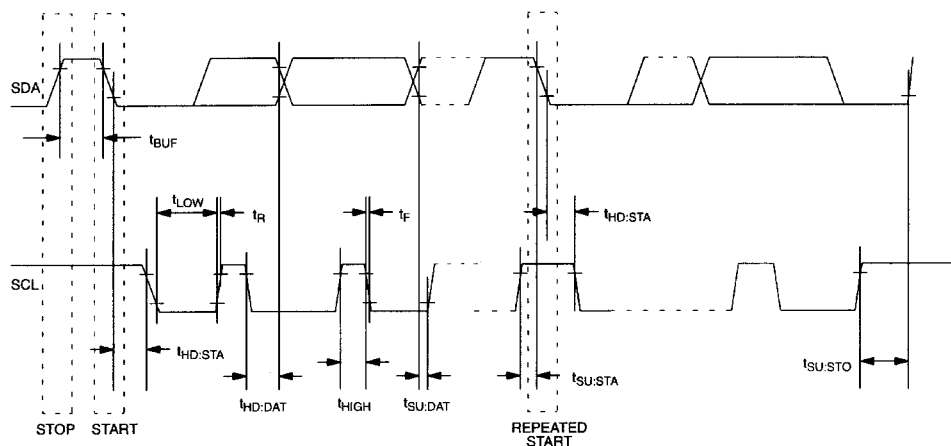
\*Unless otherwise specified.

**NOTES:**

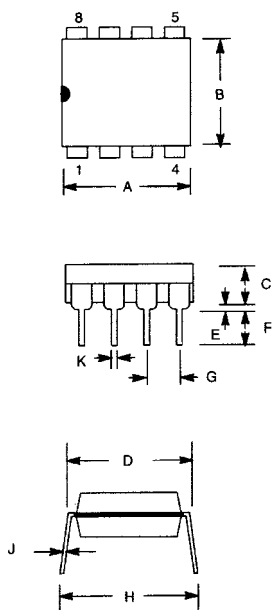
1.  $I_{CCS}$  specified with  $V_{CC} = 5.0\text{V}$  and SDA, SCL = 5.0V.
2.  $V_{CC} = 0\text{V}$ ,  $V_{BAT} = 3\text{V}$ .
3. After this period, the first clock pulse is generated.
4. A device must internally provide a hold time of at least 300ns for the SDA signal (referred to the  $V_{IHMIN}$  of the SCL signal) in order to bridge the undefined region of the falling edge of SCL.
5. The maximum  $t_{HD:DAT}$  has only to be met if the device does not stretch the LOW period ( $t_{LOW}$ ) of the SCL signal.
6.  $C_B$  – Total capacitance of one bus line in pF.
7.  $I_{CCA}$  – SCL clocking at max frequency = 100kHz.
8.  $V_{PF}$  measured at  $V_{BAT} = 3.0\text{V}$ .



### TIMING DIAGRAM Figure 8

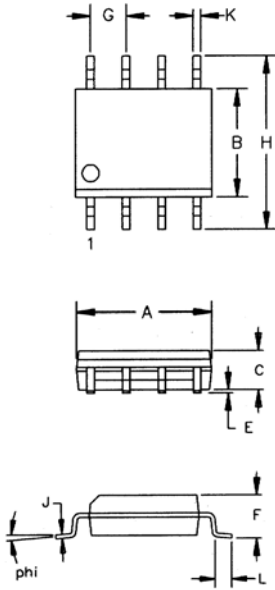


### DS1307 64 X 8 SERIAL REAL-TIME CLOCK 8-PIN DIP MECHANICAL DIMENSIONS



PKG DIM	8-PIN	
	MIN	MAX
A IN.	0.360	0.400
MM	9.14	10.16
B IN.	0.240	0.260
MM	6.10	6.60
C IN.	0.120	0.140
MM	3.05	3.56
D IN.	0.300	0.325
MM	7.62	8.26
E IN.	0.015	0.040
MM	0.38	1.02
F IN.	0.120	0.140
MM	3.04	3.56
G IN.	0.090	0.110
MM	2.29	2.79
H IN.	0.320	0.370
MM	8.13	9.40
J IN.	0.008	0.012
MM	0.20	0.30
K IN.	0.015	0.021
MM	0.38	0.53

## DS1307Z 64 X 8 SERIAL REAL-TIME CLOCK 8-PIN SOIC (150-MIL) MECHANICAL DIMENSIONS



PKG	8-PIN (150 MIL)	
	MIN	MAX
A IN.	0.188	0.196
MM	4.78	4.98
B IN.	0.150	0.158
MM	3.81	4.01
C IN.	0.048	0.062
MM	1.22	1.57
E IN.	0.004	0.010
MM	0.10	0.25
F IN.	0.053	0.069
MM	1.35	1.75
G IN.	0.050 BSC	
MM	1.27 BSC	
H IN.	0.230	0.244
MM	5.84	6.20
J IN.	0.007	0.011
MM	0.18	0.28
K IN.	0.012	0.020
MM	0.30	0.51
L IN.	0.016	0.050
MM	0.41	1.27
phi	0°	8°

56-G2008-001

# Binary-coded decimal

From Wikipedia, the free encyclopedia

In computing and electronic systems, **Binary-coded decimal (BCD)** is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of circuits needed to implement mathematical operations and a relatively inefficient encoding – 6 wasted patterns per digit. Even though the importance of BCD has diminished, it is still widely used in financial, commercial, and industrial applications.

In BCD, a digit is usually represented by four bits which, in general, represent the values/digits/characters 0-9. Other combinations are sometimes used for sign or other indications.

## Contents

- 1 Basics
- 2 BCD in electronics
- 3 Packed BCD
  - 3.1 Fixed-point packed decimal
  - 3.2 Higher-density encodings
- 4 Zoned decimal
  - 4.1 Fixed-point zone decimal
- 5 IBM and BCD
- 6 Addition With BCD
- 7 Background
- 8 Legal history
- 9 Comparison with pure binary
  - 9.1 Advantages
  - 9.2 Disadvantages
- 10 Representational variations
- 11 See also
- 12 External links
- 13 References

## Basics

To BCD-encode a decimal number using the common encoding, each decimal digit is stored in a four-bit nibble.

Decimal:	0	1	2	3	4	5	6	7	8	9
BCD:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Thus, the BCD encoding for the number 127 would be:

```
0001 0010 0111
```

Since most computers store data in eight-bit bytes, there are two common ways of storing four-bit BCD digits in those bytes:

- each digit is stored in one byte, and the other four bits are then set to all zeros, all ones (as in the EBCDIC code), or to 0011 (as in the ASCII code)
- two digits are stored in each byte.

Unlike binary encoded numbers, BCD encoded numbers can easily be displayed by mapping each of the nibbles to a different character. Converting a binary encoded number to decimal for display is much harder involving integer multiplication or divide operations. The BIOS in many PCs keeps the date and time in BCD format, probably for historical reasons (it avoided the need for binary to ASCII conversion).

## BCD in electronics

BCD is very common in electronic systems where a numeric value is to be displayed, especially in systems consisting solely of digital logic, and not containing a microprocessor. By utilising BCD, the manipulation of numerical data for display can be greatly simplified by treating each digit as a separate single sub-circuit. This matches much more closely the physical reality of display hardware—a designer might choose to use a series of separate identical 7-segment displays to build a metering circuit, for example. If the numeric quantity were stored and manipulated as pure binary, interfacing to such a display would require complex circuitry. Therefore, in cases where the calculations are relatively simple working throughout with BCD can lead to a simpler overall system than converting to 'pure' binary.

The same argument applies when hardware of this type uses an embedded microcontroller or other small processor. Often, smaller code results when representing numbers internally in BCD format, since a conversion from or to binary representation can be expensive on such limited processors. For these applications, some small processors feature BCD arithmetic modes, which assist when writing routines that manipulate BCD quantities.

## Packed BCD

A widely used variation of the two-digits-per-byte encoding is called **packed BCD** (or simply **packed decimal**), where numbers are stored with two decimal digits "packed" into one byte each, and the last digit (or nibble) is used as a sign indicator. The preferred sign values are 1100 (hex C) for positive (+) and 1101 (hex D) for negative (−); other allowed signs are 1010 (A) and 1110 (E) for positive and 1011 (B) for negative. Some implementations also provide unsigned BCD values with a sign nibble of 1111 (hex F). In packed BCD, the number +127 is represented as the bytes 00010010 01111100 (hex 12 7C), and −127 as 00010010 01111101 (hex 12 7D).

<b>Sign Digit</b>	<b>BCD 8 4 2 1</b>	<b>Sign</b>
<b>A</b>	1 0 1 0	+
<b>B</b>	1 0 1 1	−
<b>C</b>	1 1 0 0	+ (preferred)
<b>D</b>	1 1 0 1	− (preferred)
<b>E</b>	1 1 1 0	+
<b>F</b>	1 1 1 1	+ (unsigned)

Packing four-bit digits and a sign into eight-bit bytes means that an  $n$ -byte packed decimal value (where  $n$  typically ranges from 1 to 15) contains  $2n-1$  decimal digits (which is always an odd number of digits). In other words,  $d$  decimal digits require a packed decimal representation that is  $(d+1)/2$  bytes wide. For example, a four-byte packed decimal number holds seven decimal digits plus a sign, and can represent values from  $\pm 0,000,000$  to  $\pm 9,999,999$ .

While packed BCD does not make optimal use of storage (about  $1/6$  of the available memory is wasted), conversion to ASCII, EBCDIC, or the various encodings of Unicode is still trivial, as no arithmetic operations are required. The extra storage requirements are usually offset by the need for the accuracy that fixed-point decimal arithmetic provides. More dense packings of BCD exist which avoid the storage penalty and also need no arithmetic operations for common conversions.

## Fixed-point packed decimal

Fixed-point decimal numbers are supported by some programming languages (such as COBOL and PL/1), and provides an implicit decimal point in front of one of the digits. For example, a packed decimal value encoded with the bytes 12 34 56 7C represents the fixed-point value +1,234.567 when the implied decimal point is located between the 4th and 5th digits.

## Higher-density encodings

If a decimal digit requires four bits, then three decimal digits require 12 bits. However, since  $2^{10} > 10^3$ , if three decimal digits are encoded together then only 10 bits are needed. Two such encodings are *Chen-Ho encoding* and *Densely Packed Decimal*. The latter has the advantage that subsets of the encoding encode two digits in the optimal 7 bits and one digit in 4 bits, as in regular BCD.

## Zoned decimal

Some implemenatations (notably IBM mainframe systems) support **zoned decimal** numeric representations. Each decimal digit is stored in one byte, with the lower four bits encoding the digit in BCD form. The upper four bits, called the "zone" bits, are usually set to a fixed value so that the byte holds a character value corresponding to the digit. EBCDIC systems use a zone value of 1111 (hex F); this yields bytes in the range F0 to F9 (hex), which are the EBCDIC codes for the characters "0" through "9". Similarly, ASCII systems use a zone value of 0011 (hex 3), giving character codes 30 to 39 (hex).

For signed zoned decimal values, the rightmost (least significant) zone nibble holds the sign digit, which is the same set of values that are used for signed packed decimal numbers (see above). Thus a zoned decimal value encoded as the hex bytes F1 F2 D3 represents the signed decimal value  $-123$ .

## Fixed-point zone decimal

Some languages (such as COBOL and PL/1) directly support fixed-point zoned decimal values, assiging an implicit decimal point at some location between the decimal digits of a number. For example, given a six-byte signed zoned decimal value with an implied decimal point to the right of the 4th digit, the hex bytes F1 F2 F7 F9 F5 C0 represent the value +1,279.50.

## IBM and BCD

IBM used the terms **binary-coded decimal** and **BCD** for six-bit *alphanumeric* codes that represented numbers, upper-case letters and special characters. Some variation of BCD was used in most early IBM computers, including the IBM 1620, IBM 1400 series and non-Decimal Architecture members of the IBM 700/7000 series. With the introduction of System/360, IBM replaced BCD with 8-bit EBCDIC.

Bit positions in BCD were usually labelled *B*, *A*, *8*, *4*, *2* and *1*. For encoding digits, *B* and *A* were zero. The letter **A** was encoded (*B,A,I*).

In the 1620 BCD *alphanumerics* were encoded using digit pairs, with the "zone" in the even digit and the "digit" in the odd digit. Input/Output translation hardware converted between the internal digit pairs and the external standard six-bit BCD codes.

In the Decimal Architecture IBM 7070, IBM 7072, and IBM 7074 *alphanumerics* were encoded using digit pairs (using two-out-of-five code in the digits, **not** BCD) of the 10-digit word, with the "zone" in the left digit and the "digit" in the right digit. Input/Output translation hardware converted between the internal digit pairs and the external standard six-bit BCD codes.

Today, BCD is still heavily used in IBM processors and databases, such as IBM DB2.

## Addition With BCD

To perform addition in BCD, you can first add-up in binary format, and then perform the conversion to BCD afterwards. This conversion involves adding 6 to each group of four digits that has a value of greater-than 9. For example:

- $9+6=15 = [1001] + [0110] = [1111]$  in binary.

However, in BCD, we cannot have a value greater-than 9 (1001) per-nibble. To correct this, one adds 6 to that group:

- $9+6 = [0000\ 1111] + [0000\ 0110] = [0001\ 0101]$

which gives us two-nibbles, [0001] and [0101] which correspond to "1" and "5" respectively. This gives us the 15 in BCD which is the correct result.

See also Douglas Jones' Tutorial.

## Background

The binary-coded decimal scheme described in this article is the most common encoding, but there are many others. The method here can be referred to as *Simple Binary-Coded Decimal (SBCD)* or *BCD 8421*. In the headers to the table, the '8 4 2 1' indicates the four bit weights; note that in the 5<sup>th</sup> column two of the weights are negative.

The following table represents decimal digits from 0 to 9 in various BCD systems:

Digit	BCD 8 4 2 1	Excess-3 or Stibitz Code	BCD 2 4 2 1 or Aiken Code	BCD 8 4 -2 -1	IBM 702 IBM 705 IBM 7080 IBM 1401 8 4 2 1

<b>0</b>	0000	0011	0000	0000	1010	<b>Legal history</b>  In 1972, the U.S. Supreme Court overturned a lower court decision which had allowed a patent for converting BCD encoded numbers to binary on a
<b>1</b>	0001	0100	0001	0111	0001	
<b>2</b>	0010	0101	0010	0110	0010	
<b>3</b>	0011	0110	0011	0101	0011	
<b>4</b>	0100	0111	0100	0100	0100	
<b>5</b>	0101	1000	1011	1011	0101	
<b>6</b>	0110	1001	1100	1010	0110	
<b>7</b>	0111	1010	1101	1001	0111	
<b>8</b>	1000	1011	1110	1000	1000	
<b>9</b>	1001	1100	1111	1111	1001	

computer (see *Gottschalk v Benson*). This was an important case in determining the patentability of software and algorithms.

## Comparison with pure binary

### Advantages

- Scaling by a factor of 10 (or a power of 10) is simple; this is useful when a decimal scaling factor is needed to represent a non-integer quantity (e.g., in financial calculations where it is required that a computer get the same result that a human would)
- Rounding at a decimal digit boundary is easier
- Alignment of two decimal numbers (for example  $1.3 + 27.08$ ) is a simple, exact, shift
- Conversion to a character form or for display (e.g., to a text-based format such as XML, or to drive signals for a seven-segment display) is a simple per-digit mapping (conversion from pure binary involves relatively complex logic that spans digits, and gets geometrically worse as the length of the number increases).

### Disadvantages

- Some operations are more complex to implement. Adders require extra logic to cause them to wrap and generate a carry early. 15%-20% more circuitry is needed for BCD add compared to pure binary. Multiplication requires the use of algorithms that are somewhat more complex than shift-mask-add (a binary multiplication, requiring binary shifts and adds or the equivalent, per-digit or group of digits is required)
- BCD in raw form requires four bits per digit. However, when packed so that three digits are encoded in ten bits, the extra storage requirement over pure binary is insignificant for most applications.

## Representational variations

Various BCD implementations exist that employ other representations for numbers. Programmable calculators manufactured by Texas Instruments, Hewlett-Packard, and others typically employ a floating-point BCD format, typically with two or three digits for the (decimal) exponent. The extra bits of the sign digit may be used to indicate special numeric values, such as infinity, underflow/overflow, and error (a blinking display).

## See also

- Chen-Ho encoding
- Densely Packed Decimal
- Gray code

## External links

- Erik Østergaard's BCD page
- IBM: Chen-Ho encoding
- IBM: Densely Packed Decimal.

## References

- *Arithmetic Operations in Digital Computers*, R. K. Richards, 397pp, D. Van Nostrand Co., NY, 1955
- Schmid, Hermann, *Decimal computation*, ISBN 047176180X, 266pp, Wiley, 1974
- *Superoptimizer: A Look at the Smallest Program*, Henry Massalin, ACM Sigplan Notices, Vol. 22 #10 (Proceedings of the Second International Conference on Architectural support for Programming Languages and Operating Systems), pp122-126, ACM, also IEEE Computer Society Press #87CH2440-6, October 1987
- *VLSI designs for redundant binary-coded decimal addition*, Behrooz Shirazi, David Y. Y. Yun, and Chang N. Zhang, IEEE Seventh Annual International Phoenix Conference on Computers and Communications, 1988, pp52-56, IEEE, March 1988
- *Fundamentals of Digital Logic* by Brown and Vranesic, 2003
- *Modified Carry Look Ahead BCD Adder With CMOS and Reversible Logic Implementation*, Himanshu Thapliyal and Hamid R. Arabnia, Proceedings of the 2006 International Conference on Computer Design (CDES'06), ISBN 1-60132-009-4, pp64-69, CSREA Press, November 2006
- *Reversible Implementation of Densely-Packed-Decimal Converter to and from Binary-Coded-Decimal Format Using in IEEE-754R*, A. Kaivani, A. Zaker Alhosseini, S. Gorgin, and M. Fazlali, 9th International Conference on Information Technology (ICIT'06), pp273-276, IEEE, December 2006.

See also the Decimal Arithmetic Bibliography

Retrieved from "http://en.wikipedia.org/wiki/Binary-coded\_decimal"

Categories: Articles with unsourced statements since March 2007 | All articles with unsourced statements | Computer arithmetic | Numeration

---

- This page was last modified 06:46, 25 March 2007.
- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)  
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a US-registered 501 (c)(3) tax-deductible nonprofit charity.



# Mask (computing)

From Wikipedia, the free encyclopedia

In computer science, a **mask** is some data that, along with an operation, is used in order to extract information stored elsewhere.

The most common mask used, also known as a **bitmask**, extracts the status of certain bits in a binary string or number. For example, if we have the binary string `10011101` and we want to extract the status of the fifth bit counting along from the most significant bit, we would use a bitmask such as `00001000` and use the bitwise `AND` operator. Recalling that  $1 \text{ AND } 1 = 1$ , with  $0$  otherwise, we find the status of the fifth bit, since

```
10011101 AND 00001000 = 00001000
```

Likewise we can set the fifth bit by applying the mask to the data using the `OR` operator.

Similarly, we can use a *sequence* of binary numbers with a piece of data of equal length used to inform as to what parts of the data should be examined. With the bitwise operation  $(\text{NOT } X) \text{ AND } Y$ , a  $1$  in the mask ( $X$ ) instructs that the binary datum below ( $Y$ ) should be ignored, while  $0$ s in the mask ( $X$ ) tell that the data below ( $Y$ ) are to be examined. A common type of mask of this type is a subnetwork mask, which is associated with a device's IP address and used to instruct a router which bits of the address indicate the subdivision of the network the computer is on and which identify the specific computer within the subnetwork.

## Contents

- 1 Common bitmask functions
  - 1.1 Masking bits to 1
  - 1.2 Masking bits to 0
  - 1.3 Querying the status of a bit
  - 1.4 Toggling bit values
- 2 Arguments to functions
- 3 See Also

## Common bitmask functions

### Masking bits to 1

To turn certain bits on, we use the bitwise `OR` operation. Recall that  $Y \text{ OR } 1 = 1$  and  $Y \text{ OR } 0 = Y$ . Therefore, to make sure a bit is on, we `OR` it with a `1`. To leave a bit alone, we `OR` it with a `0`.

### Masking bits to 0

As we see above, there is no way to change a bit from *on* to *off* using the `OR` operation. Instead, we use bitwise `AND`. When a value is `AND`ed with a `1`, the result is simply the original value, as in:  $Y \text{ AND } 1 = Y$ . However, when we `AND` a value with `0`, we are guaranteed to get a `0` back so we can turn a bit off by `AND`ing it with `0`:  $Y \text{ AND } 0 = 0$ . To leave the other bits alone, simply `AND` them with a `1`.

## Querying the status of a bit

You can also use bitmasks to easily check the state of individual bits regardless of the other bits. To do this, you simply turn off all the other bits using the bitwise `AND` as discussed above and see if the resulting value is `0`. If it is, then the bit was off, but if the value is any other value, then the bit was on. What makes this so convenient is that you do not need to figure out what the value actually is, you just need to know that it is not `0`.

## Toggling bit values

So far we have seen how to turn bits on and turn bits off, but not both at once. What if we do not really care what the value is, we just know we want it to be the opposite of what it currently is? We can do this using the `XOR` (exclusive or) operation. `XOR` returns `1` if and only if an odd number of bits are `1`. Therefore, if two corresponding bits are `1`, the result will be a `0`, but if only one of them is `1`, the result will be `1`. Therefore we can invert the values of bits by `XOR`ing them with a `1`. If the original bit was `1`, we will get  $1 \text{ XOR } 1 = 0$ . If the original bit was `0` we will get  $0 \text{ XOR } 1 = 1$ . Also note that `XOR` masking is bit-safe, meaning it will not affect unmasked bits because  $Y \text{ XOR } 0 = Y$ , just like an `OR`.

## Arguments to functions

In programming languages such as C, bit masks are a useful way to pass a set of named boolean arguments to a function. For example, in the graphics API OpenGL, there is a command, `glClear()` which clears the screen or other buffers. It can clear up to four buffers (the color, depth, accumulation, and stencil buffers), so the API authors could have had it take four arguments. But then a call to it would look like

```
glClear(1,1,0,0); // This is not how glClear actually works and would make for unreadable code.
```

which is not very descriptive. Instead there are four defined bit fields, `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_ACCUM_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT` and `glClear()` is declared as

```
void glClear(GLbitfield mask);
```

Then a call to the function looks like this

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Internally, a function taking a bitfield like this can use logical `and` to extract the individual bits. For example, an implementation of `glClear()` might look like

```
void glClear(GLbitfield mask) {
    if (mask & GL_COLOR_BUFFER_BIT) {
        // Clear color buffer.
    }
    if (mask & GL_DEPTH_BUFFER_BIT) {
        // Clear depth buffer.
    }
    if (mask & GL_ACCUM_BUFFER_BIT) {
        // Clear accumulation buffer.
    }
    if (mask & GL_STENCIL_BUFFER_BIT) {
        // Clear stencil buffer.
    }
}
```

While elegant, in the simplest implementation this solution is not type-safe. A `GLbitfield` is simply defined to be an `unsigned int`, so the compiler would allow a meaningless call to `glClear(42)` or even `glClear(GL_POINTS)`. In C++ an alternative would be to create a class to encapsulate the set of arguments that `glClear` can accept. However, such an attempt at type safety would be at the cost of complexity.

## See Also

- Affinity mask
- Subnetwork

Retrieved from "[http://en.wikipedia.org/wiki/Mask\\_%28computing%29](http://en.wikipedia.org/wiki/Mask_%28computing%29)"

Category: Computer arithmetic

---

- This page was last modified 23:13, 28 January 2007.
- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)  
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a US-registered 501 (c)(3) tax-deductible nonprofit charity.

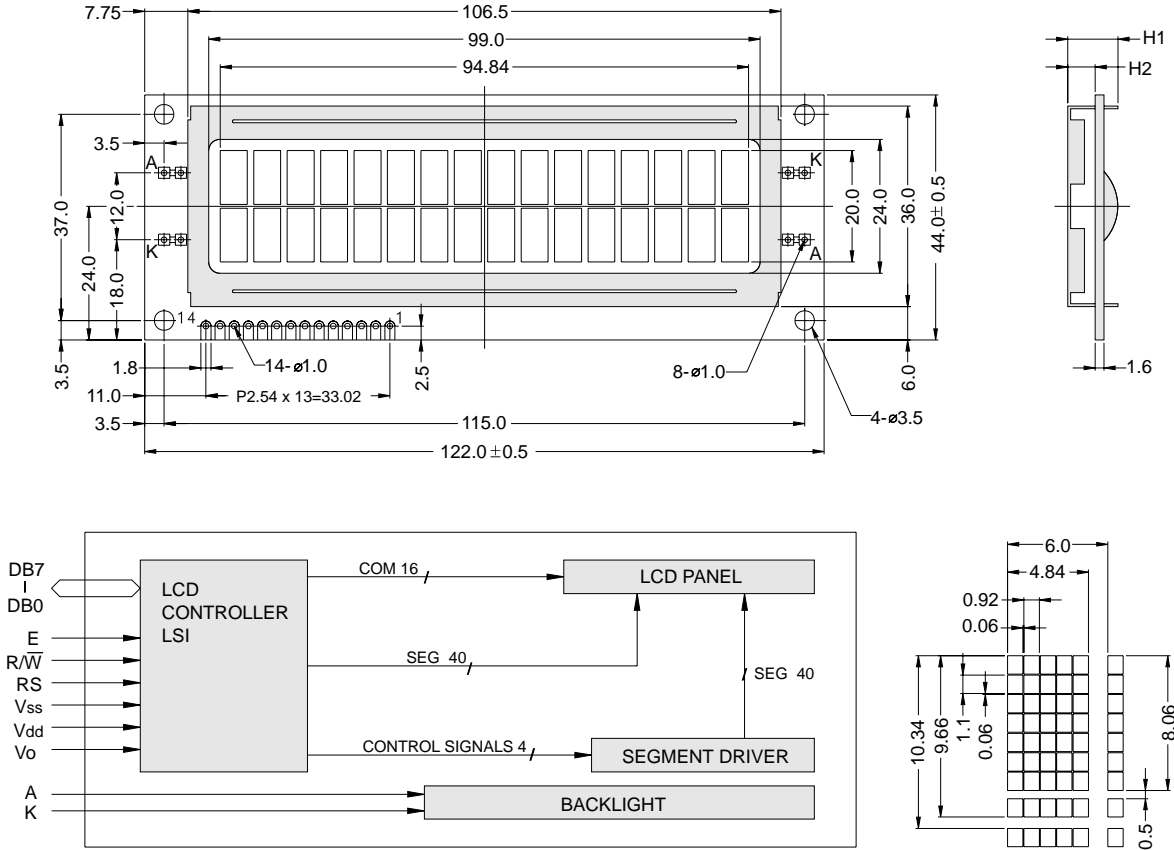
## LCD's and Accessories

Order code	Manufacturer code	Description
57-0913	n/a	16X2 ALPHANUMERIC LCD DISPLAY B/L (RC)

LCD's and Accessories	Page 1 of 2
The enclosed information is believed to be correct, Information may change 'without notice' due to product improvement. Users should ensure that the product is suitable for their use. E. & O. E.	Revision A 04/07/2003



**OUTLINE DIMENSION & BLOCK DIAGRAM**



The tolerance unless classified  $\pm 0.3\text{mm}$

MECHANICAL SPECIFICATION			
Overall Size	122.0 x 44.0	Module	H2 / H1
View Area	99.0 x 24.0	W / O B/L	4.9 / 9.0
Dot Size	0.92 x 1.10	EL B/L	4.9 / 9.0
Dot Pitch	0.98 x 1.16	LED B/L	9.4 / 13.5

PIN ASSIGNMENT		
Pin no.	Symbol	Function
1	Vss	Power supply(GND)
2	Vdd	Power supply(+)
3	Vo	Contrast Adjust
4	RS	Register select signal
5	R/W	Data read / write
6	E	Enable signal
7	DB0	Data bus line
8	DB1	Data bus line
9	DB2	Data bus line
10	DB3	Data bus line
11	DB4	Data bus line
12	DB5	Data bus line
13	DB6	Data bus line
14	DB7	Data bus line

ABSOLUTE MAXIMUM RATING									
Item	Symbol	Condition	Min.	Max.	Units				
Supply for logic voltage	Vdd-Vss	25oC	-0.3	7	V				
LCD driving supply voltage	Vdd-Vee	25oC	-0.3	13	V				
Input voltage	Vin	25oC	-0.3	Vdd+0.3	V				
ELECTRICAL CHARACTERISTICS									
Item	Symbol	Condition	Min.	Typical	Max.	Units			
Power supply voltage	Vdd-Vss	25oC	2.7	-	5.5	V			
LCD operation voltage	Vop	Top	N	W	N	W	V		
		-20oC	- 7.1	- 7.5	- 7.9	-	V		
		0oC	4.5	- 4.5	- 4.7	-	V		
		25oC	4.1	6.1	4.3	6.4	4.5	6.7	V
		50oC	4	-	4.2	-	4.4	-	V
		70oC	-	5.7	-	6	-	6.3	V
LCM current consumption (No B/L)	Idd	Vdd=5V	-	2	3	mA			
Backlight current consumption	LED/edge	VB/L=4.2V	-	120	-	mA			
	LED/array	VB/L=4.2V	-	360	-	mA			

**REMARK**

LCD option: STN, TN, FSTN

Backlight Option: LED,EL Backlight feature, other Specs not available on catalog is under request.



## Appendix B (Case Dimensioning Sketches, Program planning)

### Sheet 1

Sides 1 & 2: Rough work done when troubleshooting program around BCD realisation point.

### Sheet 2

Side 1: Early schematic (drawn when away from PC).

Side 2: Planning and calculation for breadboard program.

### Sheet 3

Sides 1 & 2: Planning for flow of breadboard program.

### Sheet 4

Sides 1 & 2: Planning for optimisation of final program to make best use of available program space and byte variables (table on side 2 was to be a plan of the multiple uses of the byte variables throughout the program however I found it easy enough to work without it, hence incomplete).

### Sheet 5

Side 1: Planning for optimisation of final program to make best use of available program space and byte variables.

### Sheet 6

Side 1: Dimensions of electronics components for use dimensioning case.

Side 2: Case dimension ideas.

### Sheet 7 - 9

Further casing ideas.

### Sheet 10

Dimensioning ideas.