Data link

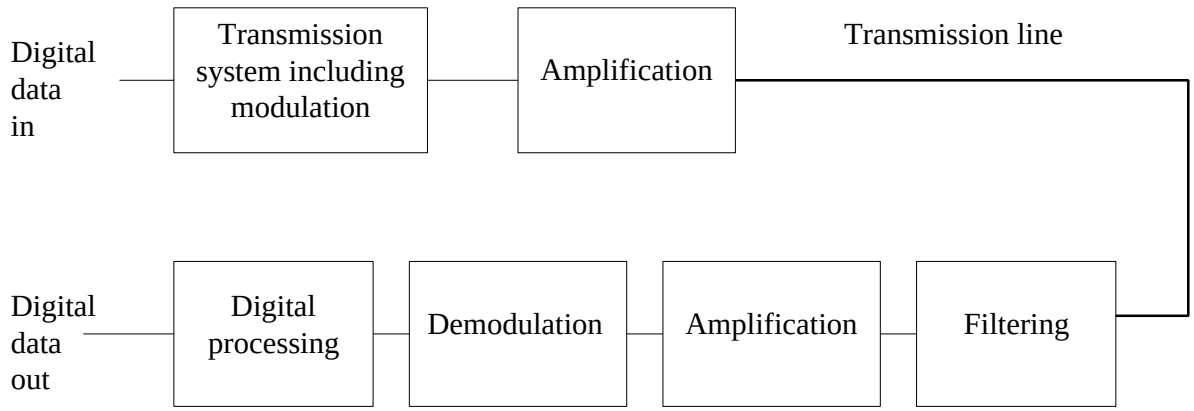
The data link forms the communications channel between the main control unit and individual appliance controllers and sensors throughout the house. Of the many possible ways this could be achieved (radio, infra-red, wires, etc.) the neatest method is through the mains wiring system, as devices being controlled are likely to use the mains as their source of power and so do not need another connection.

There is an established mains based control system called X10, but this is only designed for 110V 60Hz North American mains systems, not for European 220/240V 50Hz mains. Therefore it is not possible to produce a compatible system for European use.

The obvious problem with using mains is that there is a high current high voltage 50Hz frequency already present, so the data link has to be kept separate to prevent interference between the two signals. To do this, the most reliable way is to modulate any signal with a high frequency carrier, so that the 50Hz mains signal can be filtered out by the receiving circuit, leaving the high frequency carrier with the data modulated on it. Because of the potential dangers of using the mains directly, all the circuits were tested using a transformed down mains voltage. This was achieved using a standard transformer mounted in a sealed metal box with taps at 2, 4, 6, 8, 10 and 12V RMS. In a production system, this would either be the main power transformer for the appliance, or an additional auxiliary transformer.

There are dedicated ICs which will contain the entire transmission and receiving system, but these cannot be used here because they require a direct connection to the mains, so cannot be used with a step-down transformer.

A block diagram of the data link is shown below:

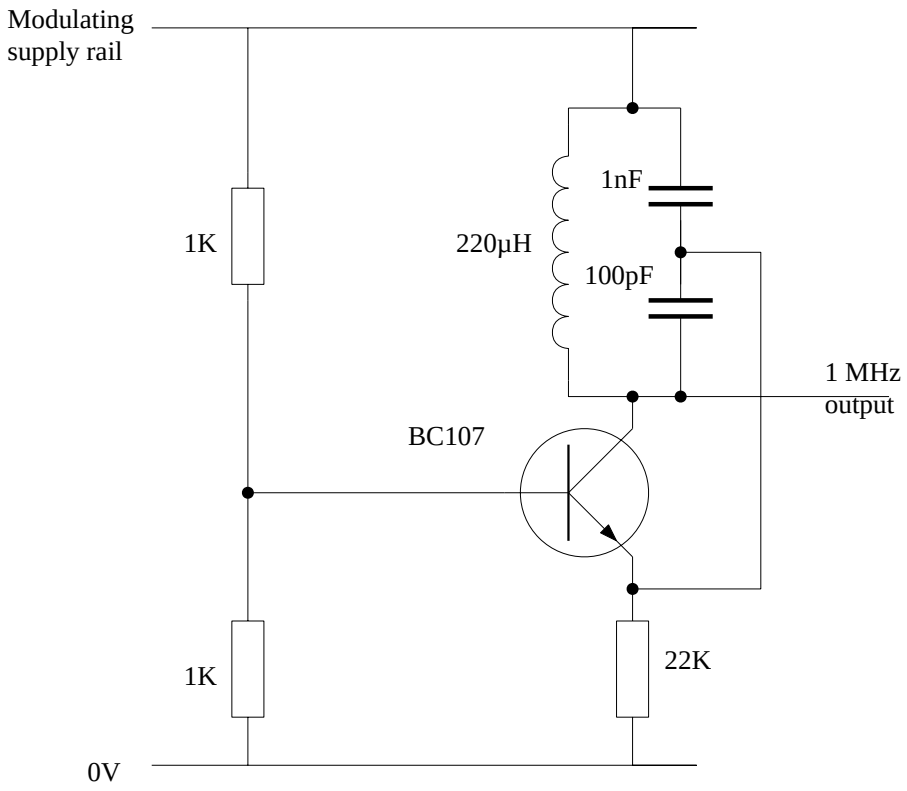


This would be repeated twice to provide a two-way (duplex) communications channel, with the same transmission line being used.

Initially only one channel was constructed to evaluate the systems being used. The receiving electronics depends on the transmitting electronics and the signal being transmitted, so the transmitting circuit was considered first.

Transmission

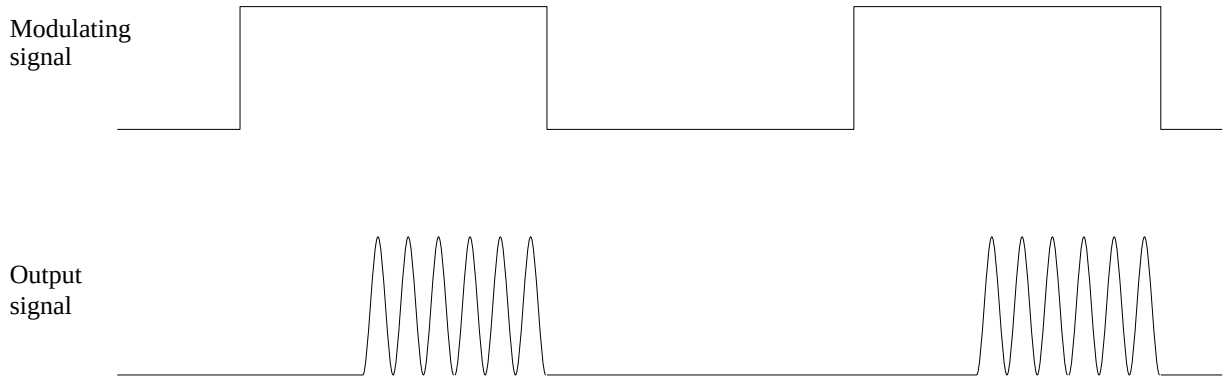
In the first instance, the following amplitude modulator (a Colpitts oscillator) was tried for the transmission system:



When this was first built, it behaved erratically, and would not oscillate properly. At first I thought this was down to the transistor, because a lack of BC107s meant a BC109 had to be used. These two transistors are similar in most respects, but they may have been operating at the limits of their specification. Examining a data book showed that the guaranteed minimum frequency a BC107 or BC109 could switch at was 150 MHz, greatly in excess of even any upper sidebands being generated here. Replacing the transistor for one from another batch or another manufacturer had no effect, ruling out the transistor as the problem.

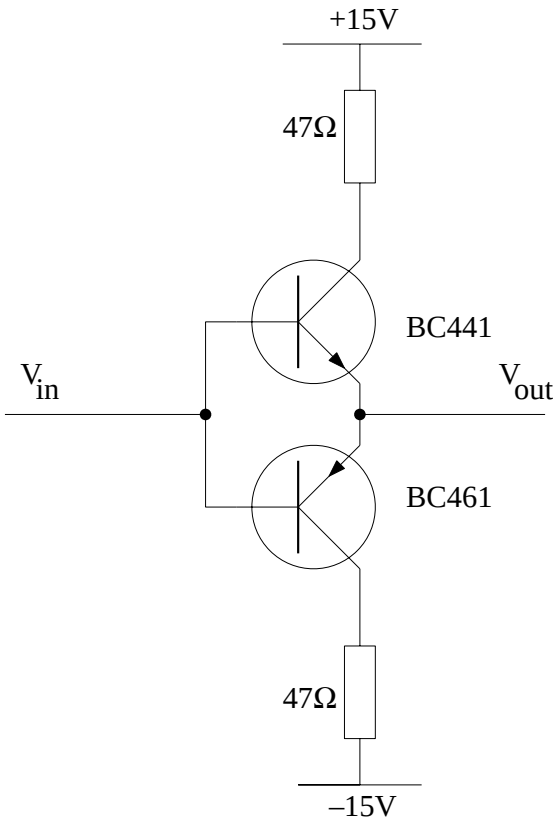
As the components were fairly close together on the breadboard, I wondered whether the capacitance of the board was having an effect. This capacitance is only slightly smaller than the 100pF capacitor being used, so it may have had an effect on it. To test this, the circuit was rebuilt spaced further apart on the board, using the same components. This did work, giving a steady output waveform reasonably close to a sine wave when given a +15V supply rail. When the supply rail was lowered, the amplitude of the output fell, but the frequency did not change noticeably.

The modulating properties of this circuit were tested by feeding it a digital square wave between 0 and 5V. The resulting bursts of sine wave showed that the output was being modulated properly. When the frequency of the modulated input was increased to around 100 kHz, the oscilloscope showed that the modulated output was taking a few microseconds to start up, so the spaces between pulses of the output increased, reducing its mark/space ratio.



This would give an upper limit on the bandwidth that could be used, but as the control information consists of only a few bytes, only a small bandwidth is required.

The problem with the system as it stands is that it has a very small power output, as any load will affect its oscillation. A high power amplifier is required to boost the signal into a form suitable for mains transmission. This was tried using a push-pull output stage as shown below:



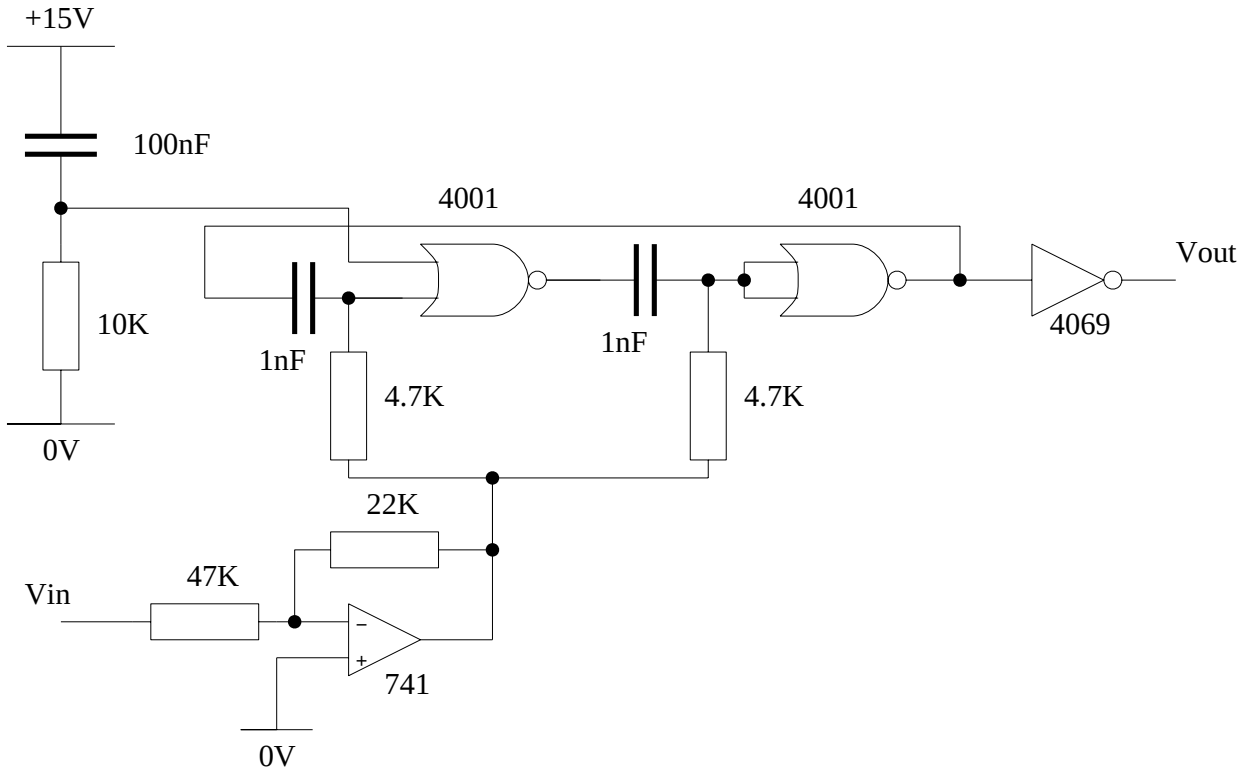
When this was connected, there was very little output at all. The transistor data book states that the maximum frequency of these transistors is 50MHz. However, this circuit is primarily designed for audio signals, so its gain may be severely reduced for such high frequency applications. The problem with using high frequency transistors is that they can supply a relatively small current – of the transistors it was possible to obtain, the 2N4427 had the highest collector current of 500mA at up to 500MHz. A semiconductor manufacturer's catalogue showed that even solid gold cased microwave transistors designed for the military could only supply 120mA. These are small compared to the 2A the BC441 and BC461 pair can supply at audio frequencies.

Although there was a very small output, the push-pull output stage was connected to a 8V AC transformer output to see if this affected the oscillator in any way. It certainly did, causing the AC signal to leak into the oscillator so that large AC signals were present throughout. This overwhelmed the small signals making it oscillate so that it was not possible to detect any 1MHz signal at all. The reason for this leakage is that bipolar transistors are being used. When the transistor is turned on, there is 0.7V across the base-emitter junction. If 11.3V (peak voltage of 8V RMS) suddenly appears at the emitter of the NPN transistor, the base will rise to 12V. Thus the AC signal will leak through the transistor with only a small amount of crossover distortion. This signal will now be present at the collector of the BC109 in the Colpitts oscillator, and so will leak through into the oscillator, preventing it from working.

There are two alternatives to prevent this problem, either filter out the sine wave or use a MOSFET, which does not suffer from the problem of the AC leaking through it. The problem with a MOSFET is that the maximum frequency is very close to that being passed through it, so problems may occur. The alternative, filtering out the 50Hz sine wave, is a better choice. This filter has to be designed so that it can filter out the 50Hz entering in one direction, while allowing the high frequency carrier to pass through in the other direction. Any active device connected directly to the output might suffer similar problems to the bipolar transistor above, so a passive filter is needed.

The problem with amplitude modulation is that it is very susceptible to noise. In a system such as the one above, very narrow bandpass filters would be required to minimise any noise from being detected as a signal. As the signal is AM, transmission factors, such as reflection caused by plugging in a new appliance to the ring main, could modify the amplitude of the signal and so generate spurious digital outputs.

To prevent these problems, a 150 kHz frequency modulator from *Electronic Systems* was tried:



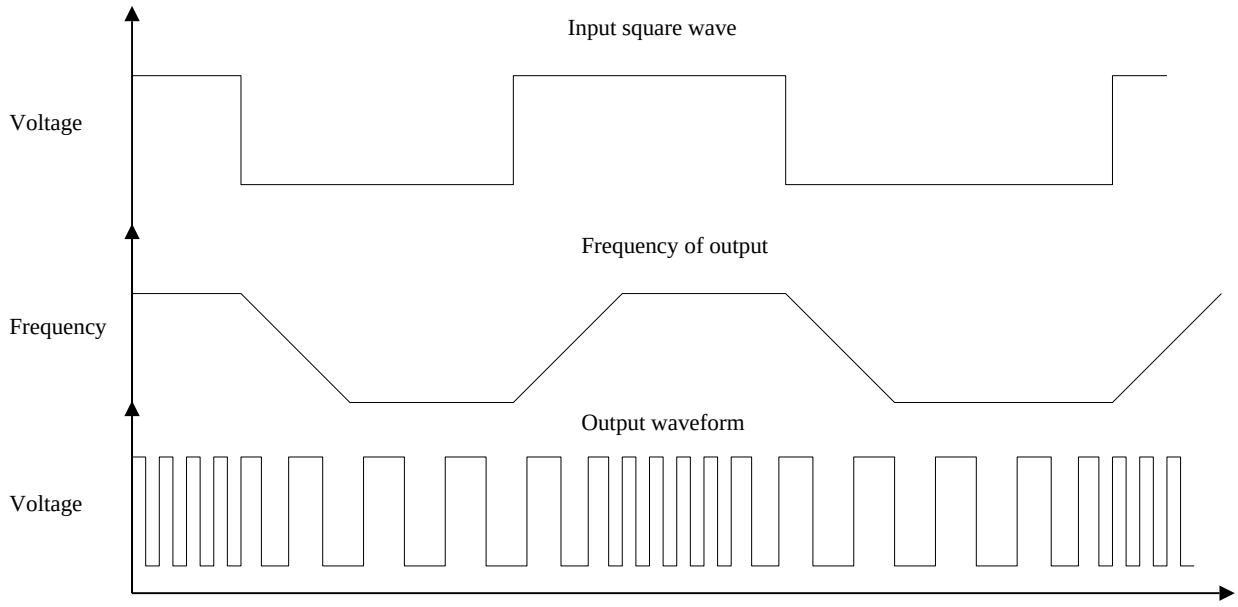
As the 741's output had a swing of ±13V, the CMOS logic was powered from +15V.

When this circuit was tried, a number of problems arose. Firstly, it was very reluctant to start oscillating. The power-on reset system composed of the 100nF capacitor and 10K resistor appeared to drop the reset signal before the components had settled down, so the system did not always start when power was applied. The situation was improved by replacing the 100nF capacitor with a 10µF value. This meant that there was a short delay before the system started oscillating, but it appeared to be more reliable.

Another problem appeared to be that the system would sometimes stop oscillating for no apparent reason. The only way to restart it was to momentarily short the 10µF capacitor, effectively performing a power-on reset. An oscilloscope showed that there was a significant amount of noise on the power supply rails. Inserting large numbers of decoupling capacitors across the supply reduced this, but the unreliability persisted. One possible explanation for it is that if a momentary drop in the power supply coincided with a high level on one of the input pins, perhaps generated by the voltage across a capacitor, then the input would be at a higher voltage than the power supply. It is known problem that if the input or output of a CMOS gate is driven beyond the supply, it will go into a state known as 'SCR latchup', where two transistors in a push-pull formation will both turn on, effectively shorting the power supply. This would result in the chip getting hot and burning itself out, but in this case the supply will still be fluctuating and will clear this problem. While this is happening the logic output of the gate could be anything, and so the circuit is quite likely to stop oscillating.

The obvious alternative is to replace the CMOS logic gates with another family. The problem is that the 4000 series is the only family which can handle power supplies above +7V. Reducing the voltage to this, or to the standard TTL level of +5V would severely limit the range of output frequencies.

When the circuit did work, it produced a reasonable square wave which did vary with the input voltage. However, it was not possible to see how rapidly the output was changing. If a square wave were input into the system, there would be a period after the edge of the square wave where the output would gradually rise or fall in frequency. The assembled circuit would not oscillate for long enough to be able to measure this, but the diagram below shows what should happen:

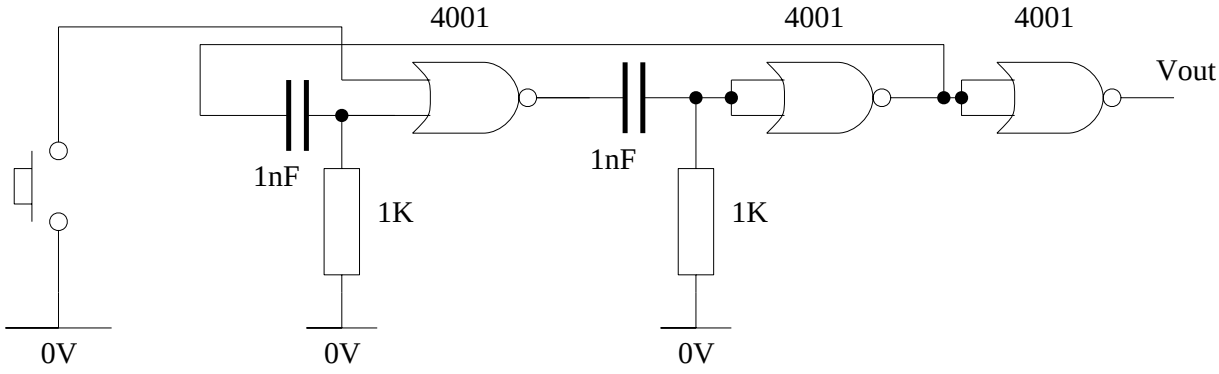


This would cause problems at the decoding end. The other problem with this circuit is that it requires a wider bandpass filter at the receiving end. This creates a larger frequency window which noise can pass through. In radio systems, the noise factor comes from factors involving the electromagnetic wave, such as diffraction and destructive interference. In a mains transmission system, these may be present, but also present are sources which generate a regular frequency of noise, such as motors or equipment containing electronic oscillators which leak into the mains. In most of these applications the fundamental frequency will be much lower than the high frequency used for transmission, but higher harmonics of the noise frequency may fall within the transmission band.

A frequency shift keyed modulator would overcome many of these problems, as the receiving circuit only needs to deal with two specific frequencies. Unfortunately, the only FSK modulator circuits I could find were essentially digital in nature, and designed for generating tones for sending down a telephone line or storing on cassette tape. This uses a 1.2 kHz signal to represent a 0 and a 2.4 kHz signal to represent a 1. Using these frequencies directly with the mains would be very susceptible to noise, and may cause electrical appliances to resonate audibly at these frequencies. The problem with using two frequencies is that the attenuation caused by the mains may be a function of the frequency. Therefore one frequency may have a greater amplitude than the other when they are received. This means some kind of automatic gain control is required. The AGC would have to be designed very carefully to ensure that both emerge at the same amplitude, but if there is no signal present the AGC will not amplify the noise on the line to make it so the same strength as the signal. The AGC would also have to adjust itself to take into account changes in signal amplitude caused by the plugging and unplugging of electrical appliances, which would affect the properties of the transmission line.

Some of these problems can be overcome if very tight timing is used on the digital signal. This method is similar in principle to Morse code, where the length of a pulse is used to determine its logic state. If a pulse is detected which has a duration not recognised as either of the two logic states, it can be ignored, providing significant noise immunity. This means one frequency can used, which simplifies the electronics immensely.

To test this, a derivative of the oscillator above was used, this time using TTL NOR gates, as the oscillator here is a fully digital system:

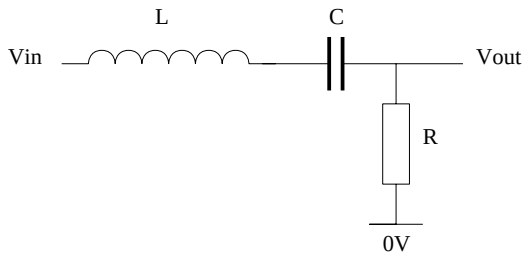


This was intended to produce a 500 kHz square wave. This was tested, and did produce a good square wave.

Transmission filtering

This square wave cannot be sent through the mains directly, as if there was a signal being sent at a higher frequency, the higher harmonics of the 500 kHz square wave might interfere with it. The square wave therefore needs to be converted to a sine wave to minimise interference, and the simplest way to do this is to filter off the higher harmonics. If a passive filter is used, it can also form the filter which prevents the mains waveform interfering with the output driver from the oscillator.

This application needs a reasonably sharp bandpass filter to filter out the 50 Hz mains frequency, as well as any mains-borne noise. The best passive filter that will perform this function is a series LC network, as shown below, as it has a small bandwidth. There is an additional benefit, as the filter has a low output impedance, a desirable property for the output.



To compensate for component tolerances, the filter must be adjustable. A variable capacitor is easier to use than a variable inductor, so this can be used as the tuning component. The highest value that was to hand was a 0-125pF AM tuning capacitor, which therefore determines the inductor to be used. At 500 kHz:

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

$$\sqrt{LC} = \frac{1}{f_0 2\pi}$$

$$L = \frac{1}{4\pi^2 C f_0^2}$$

$$L = \frac{1}{4\pi^2 .125\times10^{-12}.(500\times10^3)^2}$$

$$L = 0.81 \text{ mH}$$

This was achieved by using a 10mH and a 1mH in parallel, which gives 0.91mH. This allows the capacitor to adjust the resonant frequency around the calculated value. At the resonant frequency, a simplified estimate of the impedance *Z* is:

$$Z = 2 \times (2\pi f_0 L)$$

$$= 4\pi \times 500 \times 10^3 \times \frac{1}{\frac{1}{10}+\frac{1}{1}} \times 10^{-3}$$

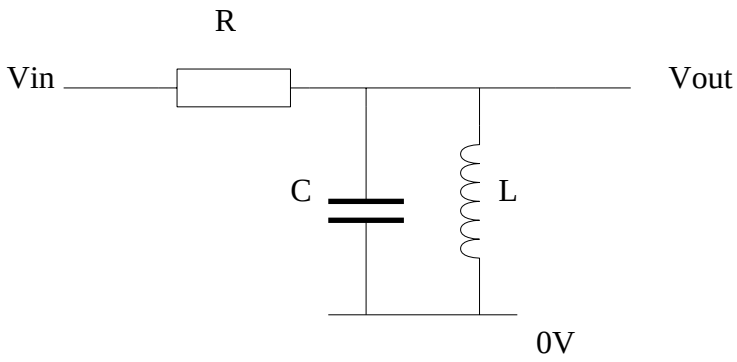
$$= 5700\ \Omega$$

The actual value is much more complicated to calculate, due to the phase difference between the inductor and capacitor. To provide a good degree of filtering, a 10K resistor was chosen as R. The output of the filter was connected to an oscilloscope, which showed that it was significantly more like a sine wave than the input. It was not possible to tune the circuit to give a perfect sine wave, but this does not matter too much, as the filter has greatly reduced the higher harmonics being produced from the output. The output from a mains transformer at 12V RMS was connected to the output of the filter, and the oscilloscope showed that very little of the 50 Hz signal was passing through to the input. This prevents the output NOR gate being affected by a large AC signal at its output.

When the oscillator was run and the AC voltage applied to the output of the filter, there was little discernible difference between the output and a pure sine wave. It is likely that the transformed mains sine wave would swamp the small high frequency output anyway.

Receiver filtering

Now a signal has been imposed on the mains, it needs to be extracted at the receiving end. The first hurdle to be overcome is to filter out the 50 Hz mains frequency. Again an LC filter is required to filter out any noise present on the mains line. A series LC filter could be used as before, except it has a low impedance, when it is better for the receiving end to have a high input impedance. The alternative is to use a parallel LC filter, which does have a high input impedance. This is shown below:



The resonant frequency is the same as before, at $1/2\pi\sqrt{LC}$. Therefore, the same values of inductor and capacitor as before can be used. Since they are in parallel, the impedance can be estimated by:

$$\frac{1}{Z} = \frac{1}{2\pi fL} + 2\pi fC$$

$$\frac{1}{Z} = \frac{1}{2\pi fL} + \frac{4\pi^2 f^2 CL}{2\pi fL}$$

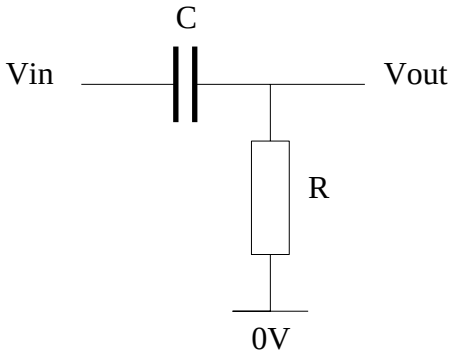
$$Z = \frac{2\pi fL}{1 + 4\pi^2 f^2 CL}$$

$$Z = \frac{2858}{1 + 1.12}$$

$$Z = 1350\ \Omega$$

Therefore, to minimise attenuation of the desired frequency while allowing a good range of attenuation for other frequencies, a 470Ω resistor was used for R.

This gave an output signal of approximately 200mV peak-to-peak, of which it could be seen that there was a considerable fluctuation due to the 50 Hz mains signal creeping through. To filter this out, another LC filter could be used. However, the variable capacitor used is bulky and expensive, and the system would need complicated tuning if more than one were used. In this situation, all that is required is to filter off the mains signal. This could be performed with a high pass active filter, but this would be operating very close to the frequency limits of the op-amp, and a filtering circuit might generate higher harmonics which the op-amp could not respond fast enough to. For these reasons, a simple passive RC high pass filter was chosen:



As the current that can be supplied by the LC filter is very small, the impedance of the RC filter needs to be as high as possible. This can be achieved by making R very large, so little current will be drawn. 100K is a suitable value. The filter will start to attenuate at the break frequency, and has a shallow increase in attenuation as the frequency decreases. To filter off the 50 Hz mains frequency, but leave a wide band for the signal, a break frequency of 100 kHz was chosen. The capacitor value can be calculated as shown below:

$$f_0 = \frac{1}{2\pi RC}$$

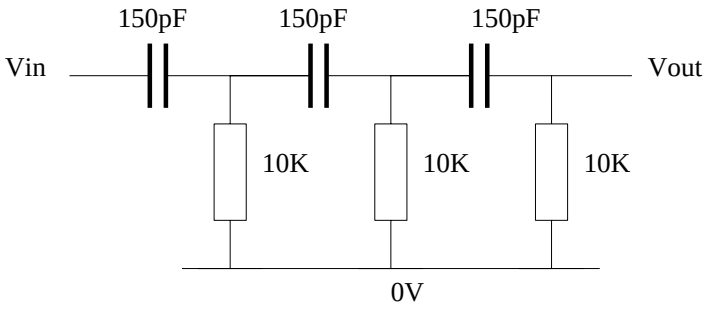
$$C = \frac{1}{2\pi fR}$$

$$= \frac{1}{2\pi \times 100\times10^{3} \times 100\times10^{3}}$$

$$= 15 \text{ pF}$$

This could be affected by board capacitance or noise, so it was decided to reduce the resistor value to 10K and increase the capacitor value to 150 pF.

When the RC filter was in place, it could be seen that it was filtering off a good proportion of the 50Hz mains signal, but not all of it. To remedy this, three RC filters were connected in series to the signal:



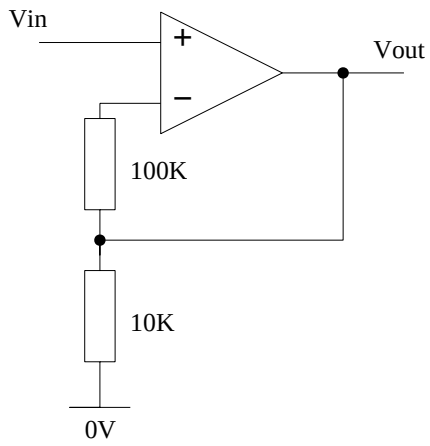
This virtually eliminated the 50 Hz interference, at the expense of the 500 kHz signal, which was reduced to around 75mV peak-to-peak in size. Therefore this will need amplifying before it is converted to a digital signal.

Receiver amplification

The 75mV 500 kHz signal needs to be amplified to make it into a manageable amplitude. There are three main types of amplifier that could be used here, op-amp, bipolar transistor or FET. Discrete transistor circuits are more complicated than op-amp based systems, and some rely on a particular type of transistor having particular characteristics. Therefore it was decided to use an op-amp amplifier.

As the frequency being used is high, the standard 741 cannot be used and so high frequency op-amps are required. As the closed loop gain of any op-amp decreases with frequency, several op-amps are necessary, each amplifying by a small amount. The OP282 and AD711 op-amps were chosen because they were able to operate at high frequencies and had high slew rates. The AD711 is a video amplifier, designed for unity gain operation at high frequencies. This can be used to buffer the signal when it has been amplified by the OP282, which is a dual op-amp.

One half of the OP282 was used in non-inverting amplifier mode to form an amplifier with gain 11:



Power supplies of ±15V were used with all op-amps. This amplifier gave reasonable performance, and seemed to be amplifying the signal properly. There was a considerable amount of distortion which showed up as blurring on the oscilloscope, but it did not appear to be significant as the main sine wave was propagating as expected. This amplifier gave an output of about 800mV peak-to-peak at the same frequency as the input.

This needs further amplification. To do this, the other half of the OP282 was used in the same non-inverting amplifier circuit as before. This circuit did not work properly, either giving an erratic output or a very small one. Adding extra decoupling to the supply rails, and replacing the OP282 chip had no effect. The input voltage to the amplifier continued to oscillate as before.

The OP282's data sheet showed what the problem was. This device has a slew rate of 8V/µs, while the amplifier circuit is trying to produce a 9V signal at 500 kHz. In one cycle the voltage goes from positive to negative to positive again, so in 1µs the output has to slew 9V. This is above that rated for the device, so this is likely to be causing the problems with the output.

The OP282 could be replaced by another type of op-amp, but it is doubtful whether cheap op-amps would be able to give the gain needed at the frequencies required.

Rethink

As well as the problems with the op-amps, the circuit would only work when one type of transformer was being used to supply the 12V AC. Other types would cause a significant drop in the transmission of the carrier signal. As the mains circuit in a house is constantly changing, with devices being switched on and off, the receiver circuit would be susceptible to these changes.

I also realised that the impedance of the transformer was causing attenuation by interfering with the filtering circuit, and effectively filtering out the carrier signal. The method of overcoming this problem that is used in baby alarms, which use the mains as a transmission medium, is to add another coil to the inductor used in the filtering circuit, which converts it into a mains transformer. As it is not possible to directly connect to the mains for the purposes of this project, it was not possible to try this.

For these reasons, I decided to abandon the mains, and modify the circuit I had built to make it transmit along a single wire link. The system was designed so that if direct mains connection were possible it could easily be modified to transmit through the mains.

Wire link transmitter

The frequency of the oscillator was reduced, so that problems with op-amp slew rates could be avoided. To minimise interference between the two channels, I decided to make one channel have a 100 kHz carrier and the other to have a 10 kHz carrier. The 10 kHz carrier could cause an audible noise if equipment resonated at that frequency, but it is likely that, if passed through the mains, it would be tiny compared to the 50 Hz signal. To ensure that the channels did not interfere with each other, the two channels were built and tested simultaneously.

The 100 kHz carrier was produced using the same oscillator circuit as before, with 4.7nF capacitors instead of 1nF capacitors. This increases the period by 4.7 times, which gives approximately 100 kHz. As with virtually all RC oscillators, it is not possible to determine the frequency precisely, but as long as any filters are tuned to match this output frequency the exact value will not matter.

The 10 kHz carrier was produced by the same principle, except with 47nF capacitors. When both of the output frequencies were displayed on an oscilloscope, they were shown to be roughly correct. Both of the enable pins on the oscillators were taken low by wire to permanently turn then on for the purposes of testing.

The same transmitter LC filter was used as before, although the inductor value had to be modified for the new frequency:

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

$$\sqrt{LC} = \frac{1}{f_0 2\pi}$$

$$L = \frac{1}{4\pi^2 C f_0^2}$$

$$L = \frac{1}{4\pi^2 .125\times10^{-12}.(10\times10^3)^2}$$

$$L = 20.2 \text{ mH}$$

The closest value to hand was a 47mH, so two were used in parallel to give 23.5mH. The impedance of this combination can be calculated as before:

$$Z = 2 \times (2\pi f_0 L)$$

$$= 4\pi \times 100\times10^3 \times 23.5\times10^{-3}$$

$$= 29500\ \Omega$$

To produce a potential divider with a reasonable range of output voltages, a 47K resistor was used as the resistor between the output and ground.

This was repeated for the other side of the link, which has to pass 10 kHz.

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

$$\sqrt{LC} = \frac{1}{f_0 2\pi}$$

$$L = \frac{1}{4\pi^2 C f_0^2}$$

$$L = \frac{1}{4\pi^2 .125\times10^{-12}.(10\times10^3)^2}$$

$$L = 2.02 \text{ H}$$

This is a very large inductance, which would be expensive to make from ready made inductors as the largest value I was able to obtain was 47mH. It would be possible to wind an inductor specially, but without any form of accurate inductance meter, its value can only be determined by trial and error. This is obviously not a very satisfactory solution.

The best compromise it was possible to find was to use four 47mH inductors in series to produce a 188mH inductance. This filters off some of the high harmonics of the 10 kHz square wave, and so prevents them interfering with the 100 kHz sine wave from the other channel. The filter does not filter off all the harmonics, so the output is not perfectly square, but this should not cause many problems.

The impedance of the inductor/capacitor combination is as follows:

$$Z = 2 \times (2\pi f_0 L)$$

$$= 4\pi \times 10\times10^3 \times 188\times10^{-3}$$

$$= 24000\ \Omega$$

Therefore a 47K resistor was used as with the 100 kHz transmitter.

The outputs of both of the transmitter circuits were joined together with a long length of single core wire, to simulate a noisy connection. In practice, the signal would be carried on screened cable, which would significantly reduce the noise problem. An oscilloscope, connected to the wire, showed that when one transmitter was activated by taking its enable input low, the output frequency was present on the transmission line. When both were triggered the output trace was blurry suggesting that the oscilloscope could not lock onto either of the frequencies. This suggests that there is a complex waveform being produced, which is difficult to detect on an analogue oscilloscope.

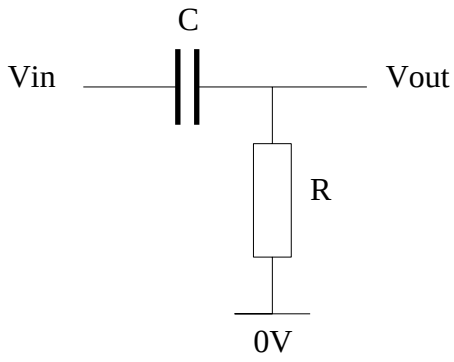
Wire link receiver

Now an transmission signal has been produced, its two component frequencies need to be separated, so that they can be detected individually. Previously, another of LC filter was used as the receive filter. This was for the mains based system, where there is a need to filter out the 50Hz sine wave, as well as any other sources of interference.

This system uses a wire link, so these extra signals should not be present. Therefore, using an LC filter is overkill when a simpler filter could be used instead. An LC filter would be required if this system were being converted to mains operation, but the whole filter would have to be redesigned anyway to incorporate a transformer which also acts as the inductor in the circuit.

The simple alternative to an LC filter is to use an RC filter. These have a much greater bandwidth than LC filters, so each section can filter off the range of frequencies containing the unwanted carrier, leaving a frequency band containing the desired carrier frequency.

For the 100 kHz receiver, a high pass RC filter is required to remove the 10 kHz carrier. At low frequencies, a capacitor has a high resistance, so needs to be at the top of a potential divider to reduce the amplitude of the low frequency signals. Therefore a high pass filter is as shown below:



The break frequency is given by $1/2\pi RC$. Before, using a 100K resistor gave a very small capacitor value. To prevent this problem, a 1K resistor was used. This means that the input impedance of the filter is reduced, but this is necessary to give a reasonable capacitor value which is less subject to noise and board capacitances. This can therefore be calculated:

$$f_0 = \frac{1}{2\pi RC}$$

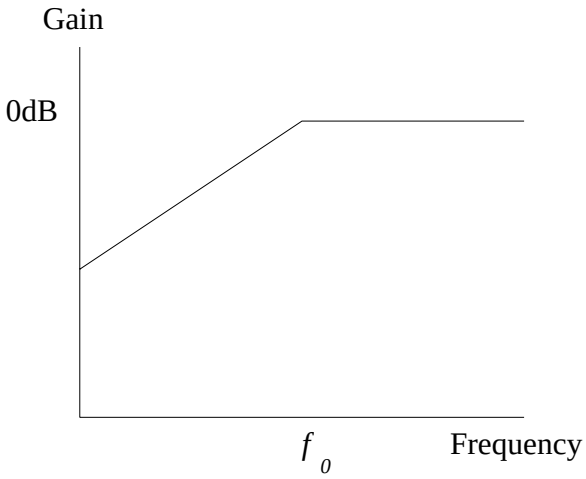
$$C = \frac{1}{2\pi fR}$$

$$= \frac{1}{2\pi \times 100\times10^3 \times 100\times10^3}$$

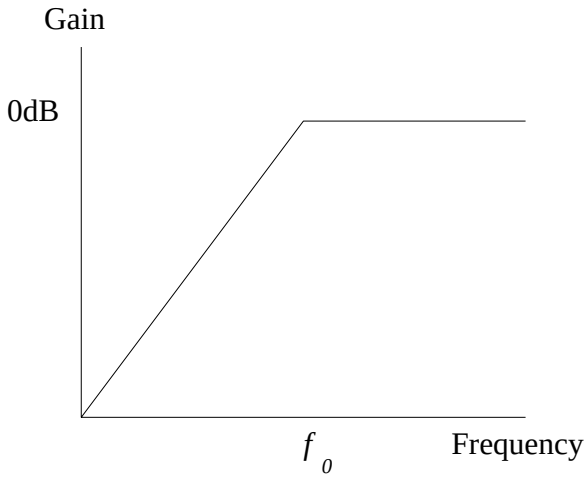
$$= 1.6 \text{ nF}$$

The nearest value to this is 1.5nF, and so this was used.

When this circuit was connected to the transmission line, its output showed that the 100 kHz signal was clearly visible, although it was blurred when the 10 kHz oscillator was activated. This problem was solved before by placing a number of RC filters in series, to increase the attenuation:



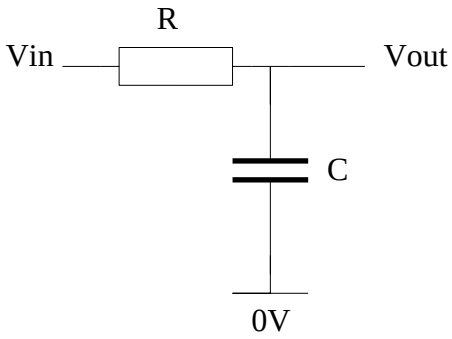
Single RC filter



Two RC filters in series

Two of these filters in series helped, but three removed most of the blurring from the signal, indicating it was almost free of low frequency carrier. The output signal was about 100mV peak-to-peak in size.

For the 10 kHz receiver, the 100 kHz carrier needs to be filtered out. This can be achieved by an RC low pass filter. This is similar to an RC high pass filter, except with the resistor and capacitor transposed:



With a high frequency input, the capacitor has a low resistance, and so pulls Vout down towards 0V, reducing the amplitude of the signal.

The break frequency of this system is again $1/2\pi RC$. As the break frequency is lower than the previous filter, higher value resistors can be used, which means that this circuit has a higher input impedance, and so loads the transmission line less. To give a break frequency of 10 kHz, 10K resistors were used. The capacitor value is then:

$$f_0 = \frac{1}{2\pi RC}$$

$$C = \frac{1}{2\pi fR}$$

$$= \frac{1}{2\pi \times 10\times10^3 \times 10\times10^3}$$

$$= 1.6 \text{ nF}$$

When this was constructed, it showed the same problem as before, namely blurring when both transmitters were activated. Putting four of these filters in series solved the problem. Presumably the problem with the output filtering from the transmitter has meant that the higher harmonics produced have increased the blurring. The output signal was around 75mV peak-to-peak in size.

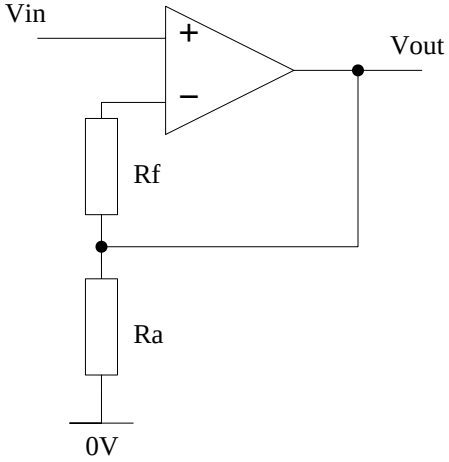
Amplification

Both of the signals need to be amplified so they can be reasonably detected. As before, there are three types of possible amplification circuit, op-amp, FET or bipolar transistor. For simplicity, I decided to use an op-amp, since the problems with high frequencies encountered before are not as important, as the transmission frequencies are lower.

To avoid problems with slew rate, the gain for each op-amp amplifier circuit should not be too high. To amplify 75mV to a detectable 7.5V requires a gain of 100, which is potentially too much for a single op-amp. I therefore decided to use at least two op-amps for each channel. Since the OP282 used before has a high slew rate and good frequency response, as well as two op-amps on one IC, this is an ideal device for this purpose.

Of the two major amplifier configurations, inverting and non-inverting, the non-inverting amplifier has an input impedance equal to that of the IC, approximately 150MΩ, while the inverting amplifier has an input impedance equal to that of its input resistor. 150M is a much greater resistance than any discrete resistor, so the non-inverting amplifier will have a higher input impedance, which is desirable as it will be drawing current from a high impedance passive filter.

The non-inverting amplifier configuration is shown below:



The gain of this amplifier is

$$Gain = 1 + \frac{Rf}{Ra}$$

Therefore, to give a gain of 10, Rf must be 9 time Ra. The precise gain does not matter, so a 100K resistor can be used for Rf, and a 10K for Ra, to give a gain of 11.

This was tried with the 100 kHz signal and found to work without any problems involving the high frequency. The other OP282 was used to amplify the 10 kHz received signal, and this also worked.

The 100 kHz signal showed some degree of blurring on the oscilloscope, after it had emerged from the amplifier. This was removed by using another 1K/1.5nF passive filter attached to the amplifier's output to filter it off. This gave a signal about 800mV in size from both of the channels. This needs further amplification to allow digital processing of the signal.

As there was a spare op-amp on each of the OP282s, it seemed sensible to use them. They were tried using the same non-inverting amplifier configuration with gain 11. This worked again, producing large (about 8V peak-to-peak), relatively clean, output signals from both channels.

Signal detection

Now we have a reasonable amplitude sine wave, this has to be converted to a logic level indicating the presence or absence of this sine wave.

There are a number of methods by which this could be achieved. A frequency-to-voltage converter could convert this to a voltage, which could then be compared with a reference voltage. This is not very satisfactory, as with no signal to converter will just pick up noise. A phase locked loop (PLL) could be constructed to detect only one precise frequency. However, PLLs are sensitive to noise, and may inadvertently lock onto the frequency from a nearby oscillator or digital circuit.

The amplitude of noise at the receiver output when the transmitter is disabled is about 1V peak-to-peak, compared with an 8V p-p signal. It might therefore be possible to smooth the signal with a capacitor, as in a power supply, to give a level depending on the level of the input. If a threshold is taken between the 1V noise and 8V signal, the two can be distinguished.

This was tested with the 100 kHz receiver. Firstly, the signal was decoupled with a 10nF capacitor to remove any DC component. This worked, but converted the signal to ±4V, the negative component of which would cause problems with a logic chip. Therefore the negative component needed to be clamped with a diode. This produced a signal between 4V and −0.7V, a signal compatible with a logic gate. The problem with this is that if the amplitude of the signal were reduced significantly, say to ±2V, it would fall outside the threshold and so not be detected. To remedy this, I tried using a pair of resistors acting as a bias network. As negligible current flows through the decoupling capacitor, these resistors were used to pull the voltage after the decoupling capacitor to around the threshold of a logic gate. CMOS logic has a threshold of about 2.5V, so 100K and 220K resistors were used to pull the logic input to:

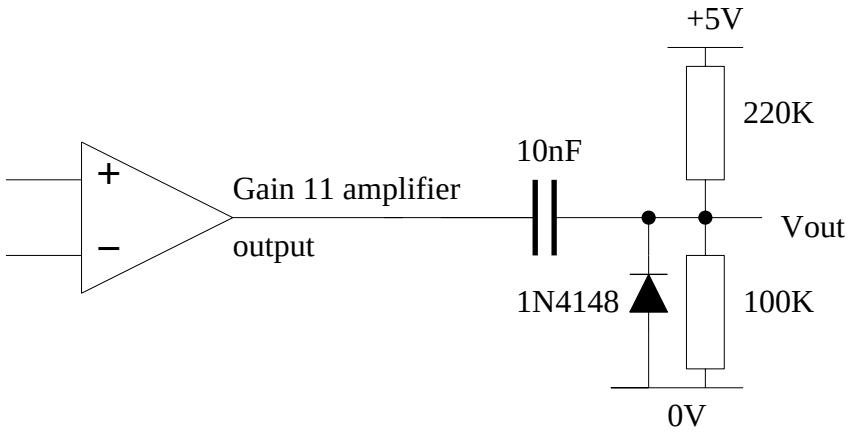
$$V = \frac{V_0 R_2}{R_1 + R_2}$$

$$= \frac{5 \times 220\times10^3}{100\times10^3 + 220\times10^3}$$

$$= 1.6\text{V}$$

This leaves about 1V margin for noise, while allowing signals as low as 1.5V to be detected.

The diagram of this section is shown below:



This produced a reasonable output signal which oscillated around 1.5V. There was one problem however. While the voltage was clamped to prevent it going below −0.7V, there was no upper limit, so it would go up to about 10V with a strong signal. This would blow any logic chip connected to Vout whcih was powered from 5V. To prevent this, the signal diode was replaced by a 5.1V zener diode. When Vout tries to go above 10V, the diode has 5.1V across it and so pulls it down, clamping it to 5.1V. If it tries to go below −0.7V, the diode conducts, and so clamps it at −0.7V.

This produced a good signal, which seemed to be ready for feeding into a logic gate. The same circuit was constructed for the 10 kHz receiver, and it appeared to be working properly.

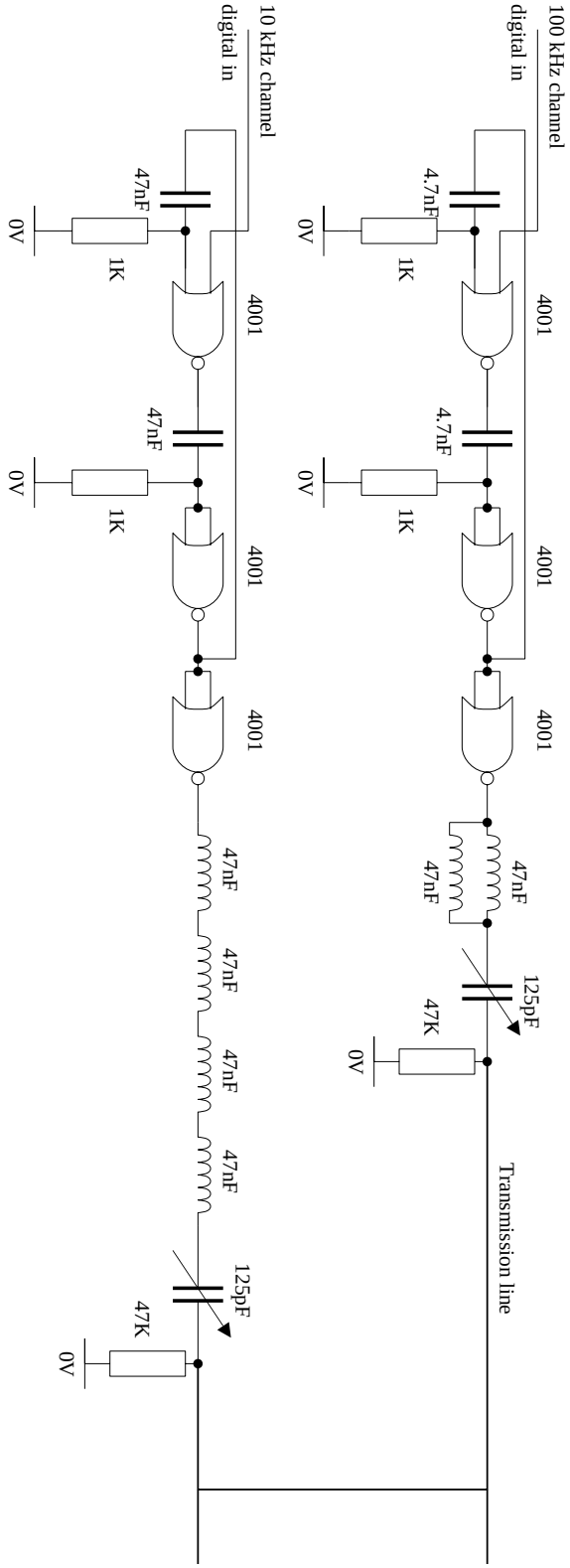
Now the signal has to be fed into a logic gate to buffer the signal. It has to be a CMOS chip, as TTL devices draw too much current, and the decoupling capacitor has a very small current supplying capability. A 4069 was tried by connecting its input to Vout of the 100 kHz receiver shown above. This caused the voltage at Vout to fall dramatically, so that the gate did not detect it. This appears to be a symptom of the device drawing too much current. It was therefore replaced with a 4001 with the inputs of each gate connected together, forming a quad inverter chip. When this was tried, the output did not show a sign of any oscillation, and stayed high. When the transmitter was turned off, the output went low. The circuit was therefore working, but without any smoothing components. This is a little strange, but it is possible that the propagation delay on the device is greater than the 100 kHz.

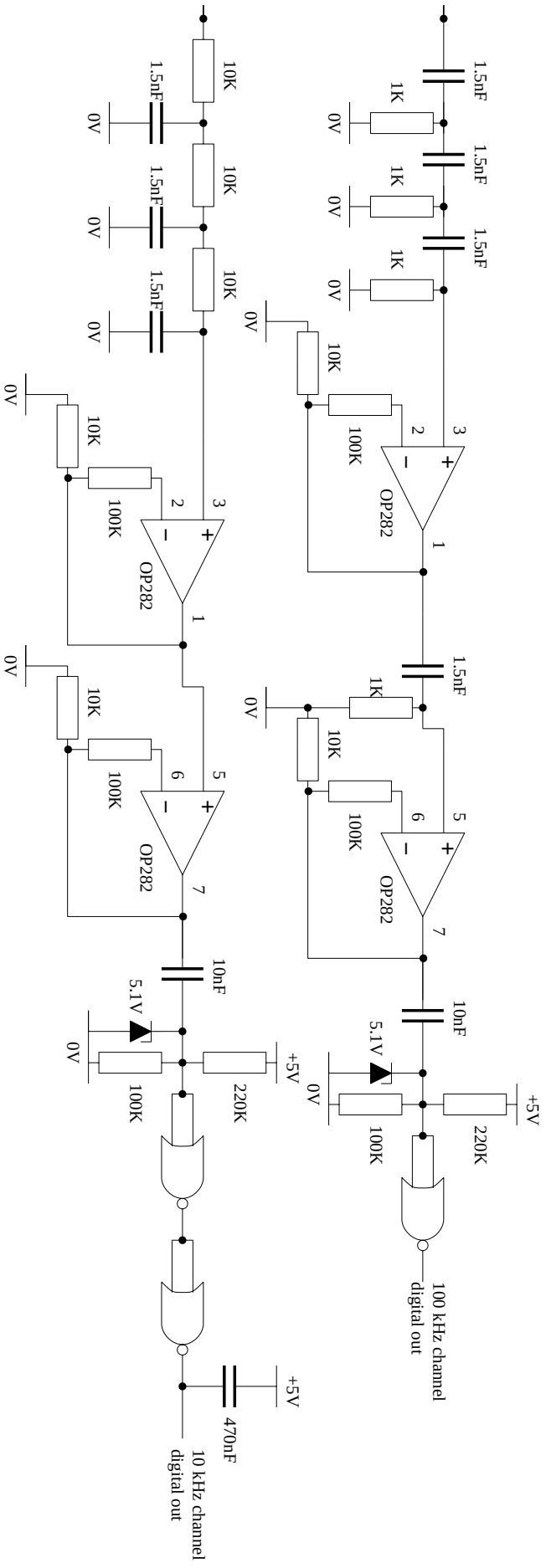
When this circuit was tried on the 10 kHz channel, the output was a noisy square wave. I tried passing it through another NOR gate inverter, which squared it up considerably. This, however, still needed conversion to a logic level by smoothing it. A number of capacitor values were tried,

but these all caused the signal to drop below the threshold of the CMOS gate. Finally, a large 470nF capacitor was tried between the gate output and +5V This did the trick, giving a reasonable high logic level when the transmitter was enabled, but not taking a long time to fall to a low logic level when the transmitter was disabled.

The data link was tested by applying a square wave to the input and examining the output waveform. More details on this are shown in the Evaluation section, but it was found that the most the 10 kHz side could manage was 80 Hz, while the 100 kHz side could handle 2500 Hz.

Data link circuit diagram





Remote control module

Now the master control unit and data link have been designed, there needs to be a system to connect to the end of the data link and communicate with the master control unit. If mains transmission were being used, this system would either be mounted in the equipment that needs to be controlled, or in a special plug which would contain the extra electronics as well as the standard fuse and cable clamp. This method would involve no modifications to equipment, and could probably be performed by the householder. Since mains transmission is not possible, the module cannot be used in this way, but it will be designed as if it can use the mains, so that the system is as close as possible to a production system.

What is required is a system to connect to the end of the wire data link, which would be the mains wiring in a production system, and control a device or devices under the commands sent through the data link. Since it would be impractical to have a separate data link wire from the master control unit for each appliance (this would be impossible with mains anyway), all the devices have to communicate along the same length of wire. It would be very difficult for each device to use a different carrier frequency on the data link, as this would severely limit the maximum number of devices possible, and would require very carefully tuned circuits, which is something that should be avoided in a production system.

The alternative is to have a simple two-way link, as has already been built, so that each device receives all the transmitted data, and then determines whether the digital data it receives is directed towards it. Such a system is similar to a computer network – the data is transmitted over a few wires to all computers, but one machine ignores anything not specifically for it.

This would require some intelligence, which it is obviously not present in a simple logic system built out logic components such as counters and shift registers. A microprocessor system similar to the master control unit could be used, but this would be vast overkill, since all the system needs to do is decode the data link signal and control a device, which does not take much processing power. In addition, the microprocessor system would be very difficult to fit into the space contained within a large mains plug.

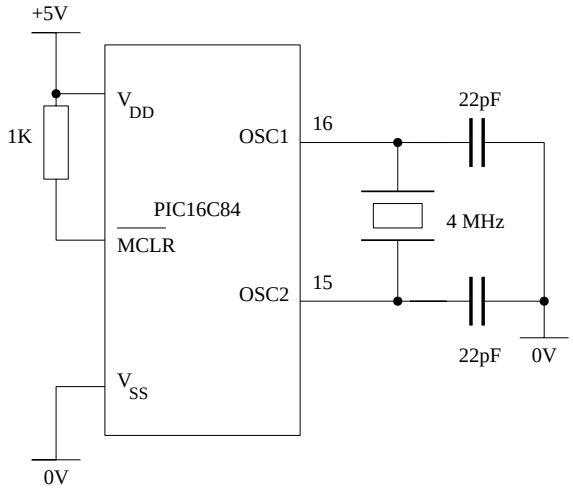
As detailed before, another possible choice is to use a microcontroller. This is essentially an entire microprocessor system on one IC, with a number of input and output ports. Some contain an internal ROM or EPROM so that the chip itself can be programmed, reducing the need for external circuitry. There are many possible devices that fit into the microcontroller category, and, like microprocessors, each family and usually each device is unique in design, so the device used has to be chosen from the start. Suitable families include the Motorola 68HC11, the ARM Thumb, the Arizona Microchip PIC, the Zilog Z8 family, the SGS SG62 family and the Intel/ Philips 8031.

Of these, the Microchip PIC seems the best choice for a number of reasons. Firstly they are extremely cheap – prices go as low as £1.89, which is not bad for the amount of computing power that a PIC provides. Secondly, the PIC itself is reasonably powerful – executing more instructions per second than many 8 or 16 bit computers. The amount of work done per instruction is less, but they still have ample power for this task. Finally, the entire system is packaged in one IC – it requires only a crystal or RC network to provide a clock, and a 2 to 6V power supply. Of the many devices in the PIC range, the PIC16C84 was chosen because it was an on-chip EEPROM (electrically erasable PROM) program memory, which means that software

development times are substantially shortened, as the chip can be erased instantly, instead of waiting for up to 20 minutes for an EPROM based microcontroller to erase under ultraviolet light. Other PIC devices have additional features such as analogue to digital converters, but the EEPROM feature is more important for development purposes. The devices in the PIC range are reasonably compatible with each other, so code initially written on the 16C84 could be transferred to other devices in the range which have these other features.

The 16C84 comes in an 18 pin plastic DIP package, which will easily fit in breadboard. For reference, the specification is shown in Appendix A, and includes a pinout diagram of the device.

The PIC was given a 5V power supply, and it /MCLR (reset) signal was pulled high with a resistor to allow it to execute a program. The PIC has an internal power-on reset, so an external one is not needed. Having done this, all that needs to be connected to it to make it function is an RC or crystal oscillator to provide the clock. Since the transmission line will carry a serial signal of some sort, an accurate timebase is essential, and this can only be provided by a crystal clock. The 16C84 is supplied in two versions, rated at 4 and 10 MHz respectively. Since this system will not be running very quickly, there is no need for the more expensive 10 MHz version. Therefore a 4 MHz crystal was used in the crystal oscillator circuit below:



A simple test program, shown below, signified that the PIC was working perfectly by producing a square wave on all the pins of the Port B input/output port. The PIC was programmed in a programmer attached to the parallel port of a computer, and then inserted into the breadboard to be tested.

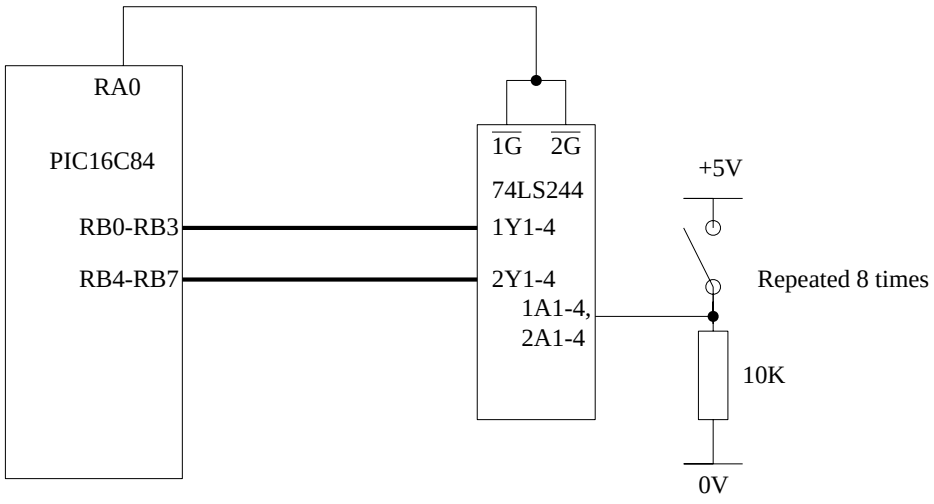
```
        LIST    P=16C84
        MOVLW   0
        TRIS    PORT_B
        OPTION
LOOP    SLEEP
        INCF    PORT_B,F
        GOTO    LOOP
        END
```

PIC hardware

In addition to the simple system support hardware, extra hardware is required for this particular application. This needs to be connected to the 13 input/output pins provided by the 16C84, which are composed of a 5 bit port (Port A) and an 8 bit port (Port B).

Firstly, a means is require to allow each device attached to the data link to be able to recognise that a command is for it. One possibility is to hard-wire an identification code into each PIC, so that each device is unique. This can be done either in the main program memory or in a special set of ID locations set aside for this purpose. The problem with this is that the devices could not be mass produced very easily, as each one would be a slightly different. It also means the code cannot be changed after production.

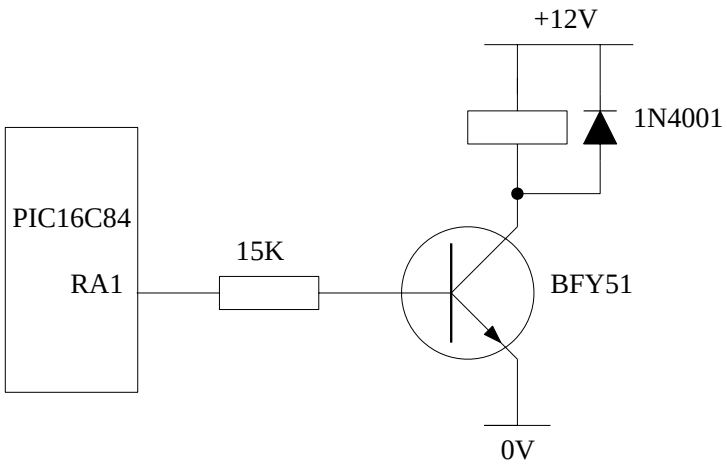
An alternative is to use a hardware method, where the PIC reads the code from external hardware through an input port. The obvious way to do this is to use an 8 way DIP switch, providing 256 different combinations. If this were connected directly to an input port, it would occupy 8 of the 13 I/O lines, which is not ideal. A better idea is to treat the 8 bit output port like a processor bus, and only connect the DIP switch to the 8 bit port when it is required. This can be performed with a tristate. Since the 74LS244 was used before, there is no reason why not to use it again. The 244 has two active low enable pins, one controlling a block of 4 bits. If these are connected together and connected to an output pin, the 8 bit DIP switch output can be connected to Port B only when the output pin is low. For this pin, pin RA0 on Port A was chosen, as it is a general purpose pin. The diagram of the DIP switch system is shown below:



This was tested, and did work. The switches were set to binary 7, which is the code used to signify the immersion heater in the master control software (see later).

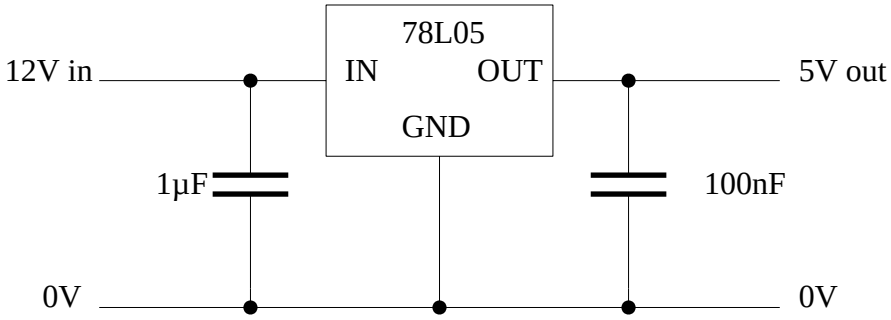
The other major hardware that is required is a means of controlling the electrical device itself. If it were a resistive or inductive device, a triac could be used to give proportional control, but this cannot be done without direct access to the mains. A system designed to work on low voltage AC would require considerable modification to cope with mains voltages. The alternative is to use a relay for simple on/off control. This isolates the high voltages, and the system will be able to switch any voltage up to the maximum rating of the relay. A production system would require proportional control, an this could be managed with a triac, but a relay will suffice for the purposes of this project.

A relay coil requires a significant amount of current, more than a PIC output pin could supply. To be able to switch large currents, large relays generally require 12V, which could not be provided by the PIC anyway. Therefore, a buffer is needed between the PIC output pin and the relay. There are many ICs which could perform this function, but this is overkill since only one output is needed. The simplest method to do this is to use a transistor. A BFY51 NPN transistor can handle the current drawn by a relay, and has enough current gain to be driven by a PIC output pin. It was tried in the standard configuration shown below, including a protection diode across the relay coil:



This worked quite well, and was able to be powered by the output current from a PIC pin.

To enable the PIC and relay to be powered from a single power supply, the unit needs a voltage regulator to convert 12V to 5V. This simplest regulator available is a single 78L05 device, which only requires two decoupling capacitors to perform this function, and can supply 100mA. One was connected as shown below, and found to work:



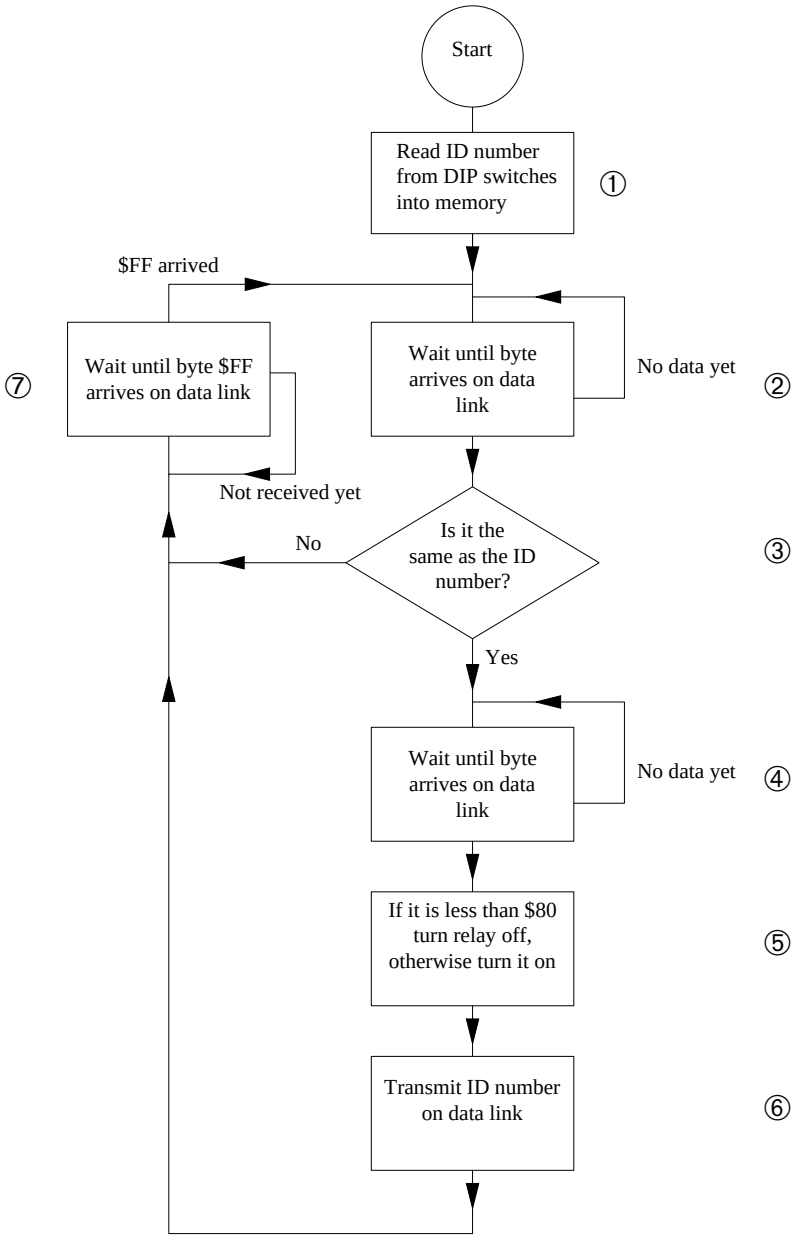
The only hardware left to interface with is the data link itself. The serial interfacing code (see next section for further details) needs to use RA4 as the receive pin, and can use any pin as the transmit pin. Since RA2 is not being used, it seems sensible to use this. The data link has a digital input and a digital output, so they were connected directly to these I/O pins. They could not be tested without the full serial interfacing code being present.

Remote module software

The PIC in the remote module needs to be programmed to accept commands from the data link and then decode them. Because this is mainly a hardware project, the software will not be examined in too much detail, and so only a basic outline of it is presented here, as well as the full source code to allow it to be inspected more thoroughly if desired.

The data link is a serial system, so a serial data stream has to be sent along it. Performing serial communications in software is a common task for a PIC, so the manufacturers, Arizona Microchip, have written a software routine to do this in Application Note AN555, which is freely distributable. Since there is no point in rewriting the code myself, I have included a slightly modified version of this code in the module software. This is clearly marked so it is possible to determine the code I have written.

A flow chart for the software is shown below. The numbers enable sections in the source code to be linked with this diagram.



Essentially, the module listens for an ID code. If it is not its own, it waits until the end of the data packet. If it is, it controls the device and sends a byte back to the master control unit, so it knows that the device has responded. It then waits for the end of the data packet. This allows variable length data packets, so that, for example, an oven could be controlled by sending more data bytes which set the temperature, control whether the fan is turned on, and so on, while this would be ignored by all other devices.

The code is split into five chunks, four of which (are related to the serial link, and are direct copies from Application Note AN555. The main control code for the remote control unit is contained in MAIN.ASM, which is shown below. The other four files, which are copied from AN555, are not shown here. Essentially they provide the routines GetChar and PutChar and their support software.

Main routines

```
     TITLE           “Home Automation System Remote Control Unit”
     SUBTITLE        “Basic on/off control program, version 1.00”


          Processor       16C84
          Radix   DEC
          EXPAND

          include         “16Cxx.h”

; set up RS232 serial link

_ClkIn          equ     4000000         ; Input Clock Frequency is 4 Mhz
_BaudRate       set     75              ; Baud Rate (bits per second) is 75
_DataBits       set     8               ; 8 bit data, can be 1 to 8
_StopBits       set     1               ; 1 Stop Bit, 2 Stop Bits is not
implemented

#define _PARITY_ENABLE  FALSE           ; NO Parity
#define _ODD_PARITY     FALSE           ; EVEN Parity, if Parity enabled
#define _USE_RTSCTS     FALSE           ; NO Hardware Handshaking is Used

          include         “rs232.h”

; settings for remote unit

id_value      equ      0x20              ; reserve memory location to
                                         ; store ID value
#define       id_latch  _porta,0         ; pin connected to 74LS244 enable
#define       relay     _porta,1         ; pin to use for relay


     ORG     _ResetVector
     goto    Start


     ORG     _IntVector
     goto    Interrupt
```

```
;    Main program

Start:
     movlw     0xFE          ; make port A all inputs except RA0
     movwf     _trisa
     movlw     0xFF          ; make port B all inputs
     movwf     _trisb

     movlw     0x00          ; enable 74LS244, reset rest of port ①
     movwf     _porta
     movf      _portb,W      ; read port B value into W register
     movwf     id_value      ; store in memory
     movlw     0x01          ; disable 74LS244
     movwf     _porta

     call    InitSerialPort  ; set up serial port

NewCommand:

     call      WaitForNext   ; get a byte from data link          ②
     movf      RxReg,w       ; move byte into W

     subwf     id_value,W    ; subtract W from ID, put in W       ③
     btfsc     _z            ; is Z set (bytes are equal => ID is ours)
     goto      CommandForUs  ; if so, receive command

                             ; if not, wait until $FF is received
WaitForEnd:

     call      WaitForNext   ; get a byte from data link          ⑦
     movf      RxReg,w       ; move byte into W
     sublw     0xFF          ; subtract $FF from received byte
     btfss     _z            ; is Z clear (not $FF)
     goto      WaitForEnd    ; if not carry on waiting

     goto      NewCommand    ; if so, wait for a new command

CommandForUs:

     call      WaitForNext   ; get a byte from data link          ④
     btfss     RxReg,7       ; is it above $7F (bit 7 set)
     bcf       relay         ; if not turn off relay

     btfsc     RxReg,7       ; is it above $7F (bit 7 set)      ⑤
     bsf       relay         ; if so, turn on relay

     movf      id_value,W    ; read ID value
     movwf     TxReg         ; put it transmit register

     call      PutChar       ; send it                          ⑥

CommandSendingAck:
     btfsc     _txmtProgress ; has it finished?
     goto      CommandSendingAck  ; if not, wait until it has
                             ; - could do other jobs here
     goto WaitForEnd         ; if so, wait for end of data packet
```

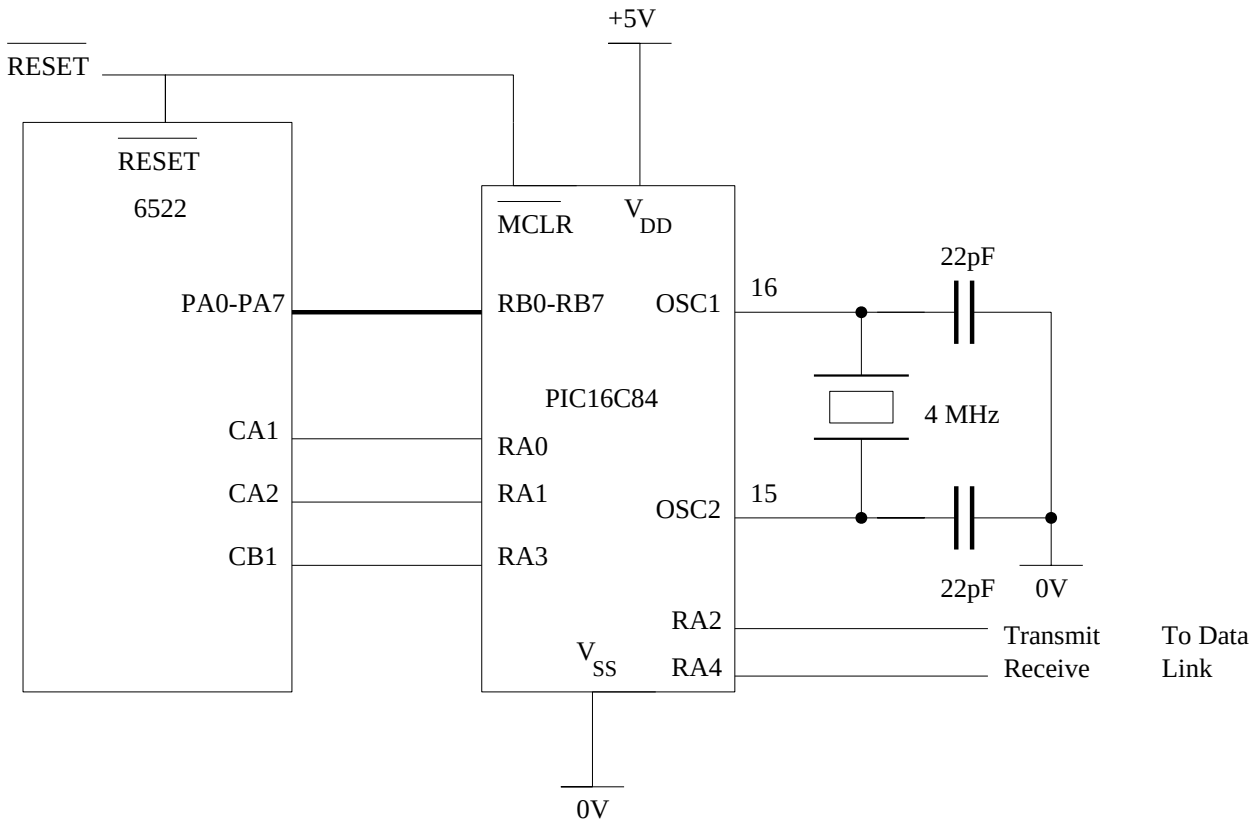
```
; routine to wait for next byte to be received

WaitForNext:
     call    GetChar         ; wait for a byte reception
     btfsc   _rcvOver        ; _rcvOver Gets Cleared when a Byte Is
                             ; Received (in RxReg)
     goto    WaitForNext     ; USER can perform other jobs here,
                             ; can poll _rcvOver bit
     return

; ******* Code below this point is copied from AN555 ********

;***********************************************************************;
                             RS-232 Routines
;;**********************************************************************;
                             Interrupt Service Routine
;
; Only RTCC Inturrupt Is used. RTCC Inturrupt is used as timing for Serial
; Port Receive & Transmit
; Since RS-232 is implemented only as a Half Duplex System, The RTCC is
; shared by both Receive & Transmit Modules.
;       Transmission :
;                       RTCC is setup for Internal Clock increments and
;                       interrupt is generated when
;                       RTCC overflows. Prescaler is assigned, depending on
;                       The INPUT CLOCK & the
;                       desired BAUD RATE.
;       Reception :
;                       When put in receive mode, RTCC is setup for external
;                       clock mode (FALLING EDGE)
;                       and preloaded with 0xFF. When a Falling Edge is
;                       detected on RTCC Pin, RTCC
;                       rolls over and an Interrupt is generated (thus Start
;                       Bit Detect). Once the start
;                       bit is detected, RTCC is changed to INTERNAL CLOCK
;                       mode and RTCC is preloaded
;                       with a certain value for regular timing interrupts to
;                       Poll RTCC Pin (i.e RX pin).
;
;***********************************************************************
Interrupt:
     btfss   _rtif
     retfie                          ; other interrupt, simply return &
                                     ; enable GIE
;
; Save Status On INT : WREG & STATUS Regs
;
     movwf   SaveWReg
     swapf   _status,w               ; affects no STATUS bits : Only way OUT
                                     ;to save STATUS Reg ?????
     movwf   SaveStatus
;
     btfsc   _txmtProgress
     goto    _TxmtNextBit            ; Txmt Next Bit
     btfsc   _rcvProgress
     goto    _RcvNextBit             ; Receive Next Bit
     goto    _SBitDetected           ; Must be start Bit
;
RestoreIntStatus:
     swapf   SaveStatus,w
```

```
     movwf   _status                ; restore STATUS Reg
     swapf   SaveWReg, F            ; save WREG
     swapf   SaveWReg,w             ; restore WREG
     bcf     _rtif
     retfie

;
;*************************************************************************;
;
;
; Configure TX Pin as output, make sure TX Pin Comes up in high state on
; Reset
; Configure, RX_Pin (RTCC pin) as Input, which is used to poll data on
; reception
;
;  Program Memory :     9 locations
;  Cycles         :     10
;*************************************************************************

InitSerialPort:
     clrf    SerialStatus

     bcf     _rp0                          ; select Page 0 for Port Access
     bsf     TX                            ; make sure TX Pin is high on
                                           ; powerup, use RB Port Pullup
     bsf     _rp0                          ; Select Page 1 for TrisB access
     bcf     TX                            ; set TX Pin As Output Pin, by
                                           ; modifying TRIS
  if _USE_RTSCTS
     bcf     _RTS                          ; RTS is output signal,
                                           ; controlled by PIC16Cxx
     bsf     _CTS                          ; CTS is Input signal, controlled
                                           ; by the host
  endif
     bsf     RX_Pin                        ; set RX Pin As Input for
                                           ; reception
     return
;
;*************************************************************************
     include “txmtr.asm”             ; The Transmit routines are in file
                                     ; “txmtr.asm”
     include “rcvr.asm”              ; The Receiver Routines are in File
                                     ; “rcvr.asm”

;*************************************************************************
     END
```

Master control transmitter

Since the master control unit has to send data along the transmission line at a very slow 75 bits per second, it will spend a long time doing this, which will slow the system down. As PICs are very cheap and simple, it seems sensible to dedicate another PIC as a transmitter, which leaves the master control unit free to get on with other things while transmission is taking place. The intelligence provided by the PICs also allows additional features such as error correction to be programmed into them, which makes the data link more robust.

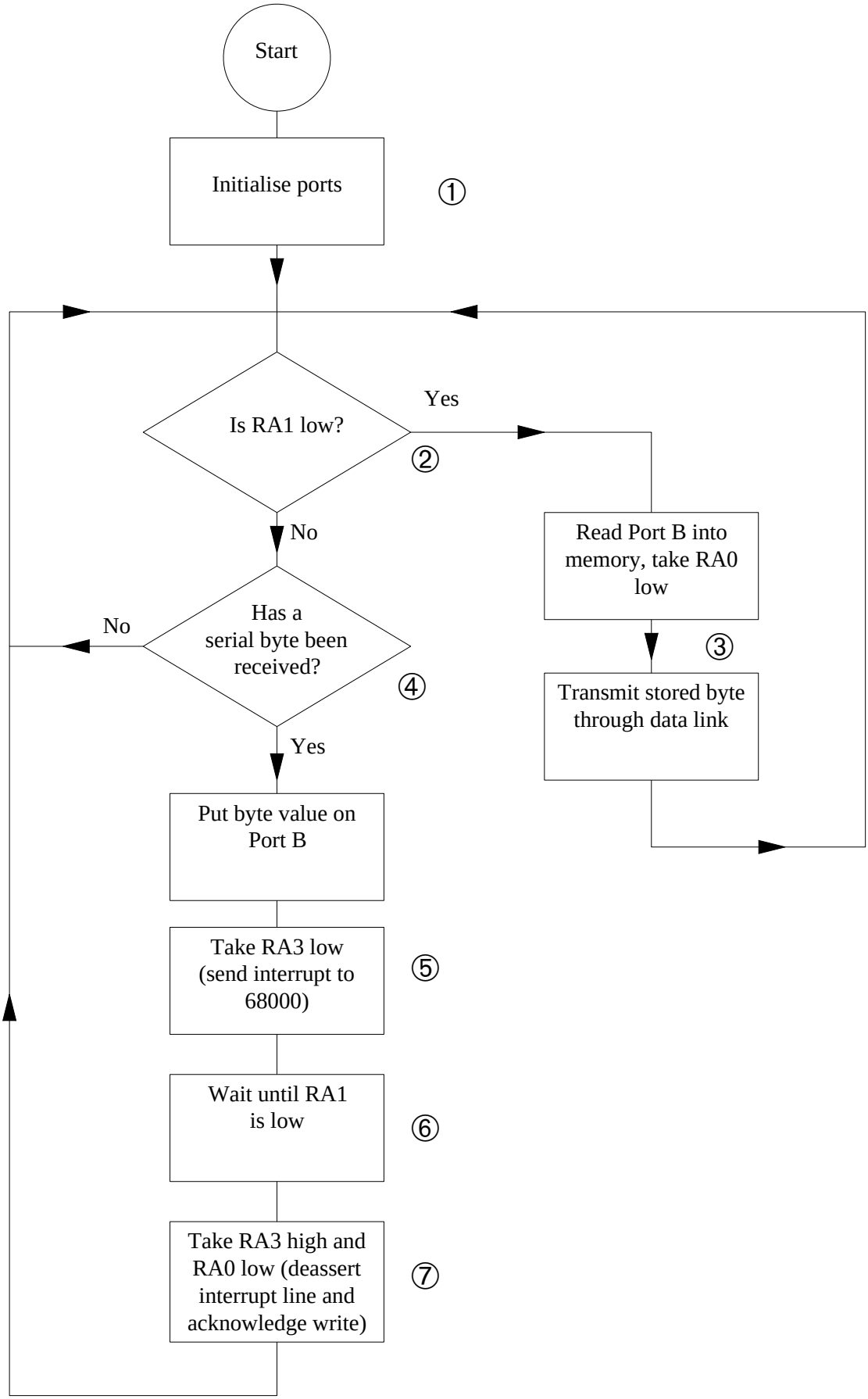
The power and crystal connections are similar to before, but this time the PIC needs to be connected directly to the main control unit. The most obvious place for this is one of the 6522's input/output ports, which allows parallel data transmission to and from the PIC. Since the 6522's Port A allows automatic read and write handshaking, which means the port will automatically generate a data strobe and an acknowledge signal when talking to another device, this seems the better port to use, and can be connected to the PIC's Port B. PIC I/O lines RA0 and RA1 can be used as handshaking lines with the 6522's to control data transfer on the bus. This leaves RA4, which must be the serial receive line, and RA2, which can be the transmit line as before. The last line, RA3, can be used to generate an interrupt when a serial byte is received by being connected to the 6522's CB1. The circuit diagram below shows these connections:



The reset line goes to the control unit's master /RESET signal, so that everything is reset at the same time.

This could not be tested without the software, which is detailed below. It is similar to that of the remote module, as it uses the same serial transmit/receive code, but employed in a different situation, so the main controlling code is different.

The software is outlined in the flowchart below:



The listing is shown below. It uses the same serial files as before, and, to save space, the second half of the source file has been omitted, as it contains the same code as in the remote unit's code listing. The omitted section is headed RS232 Routines in that listing.

```
     TITLE          "Home Automation System Master Control Unit"
     SUBTITLE       "Data link transceiver, version 1.00"


          Processor       16C84
          Radix   DEC
          EXPAND

          include         "16Cxx.h"

; set up RS232 serial link

_ClkIn          equ     4000000          ; Input Clock Frequency is 4 Mhz
_BaudRate       set     75               ; Baud Rate (bits per second) is 75
_DataBits       set     8                ; 8 bit data, can be 1 to 8
_StopBits       set     1                ; 1 Stop Bit, 2 Stop Bits is not
implemented

#define _PARITY_ENABLE  FALSE            ; NO Parity
#define _ODD_PARITY     FALSE            ; EVEN Parity, if Parity enabled
#define _USE_RTSCTS     FALSE            ; NO Hardware Handshaking is Used

          include         "rs232.h"

; settings for transceiver unit

#define       CA1  _porta,0              ; define 6522 pins
#define       CA2  _porta,1
#define       CB1  _porta,2


     ORG     _ResetVector
     goto    Start
;

     ORG     _IntVector
     goto    Interrupt
;
;    Main program

Start:

     movlw     0xFF                     ; make port B inputs        ①
     movwf     _trisb
     movlw     0xFF                     ; make all Port A high (otherwise
     movwf     _porta                   ; may cause 68K IRQ when TRIS set)
     movlw     10110b                   ; make RA3 (CB1) and RA0 (CA1)
     movwf     _trisa                   ; outputs, rest inputs
     call      InitSerialPort           ; set up serial port
```

```
CheckSend:
     btfss     CA2                         ; is CA2 low (byte sent)?
     goto SendTxdByte                      ; if so, transmit 6522 byte  ②

     call    GetChar                       ; wait for a byte reception
     btfsc   _rcvOver                      ; if nothing has been received
     goto CheckSend                        ; carry on waiting           ④

                                           ; if something has, read it
GetRxdByte:
     movf RxReg,w                          ; move received byte into W
     movwf     _portb                      ; and then to port B         ⑤
     bcf  CB1                              ; assert CB1 (cause 68K IRQ)

WaitFor6522Ack:

     btfsc     CA2                         ; wait for 6522 reply        ⑥
     goto      WaitFor6522Ack              ; not yet? Carry on waiting..

     bsf  CB1                              ; assert CB1 (clear IRQ)
     bcf  CA1                              ; reply to write             ⑦
     nop                                   ; make sure 6522 has got it
     nop
     nop
     nop
     nop
     nop
     bsf  CA1                              ; and deassert CA1
     goto CheckSend                        ; and now wait for next

SendTxdByte:

     movf _portb,W                         ; read value on Port B       ③
     movwf     TxReg                       ; get ready to transmit
     bcf  CA1                              ; signify data has been taken
     call    PutChar                       ; send it
SendingTxdByte:
     btfsc   _txmtProgress                 ; has it finished?
     goto    SendingTxdByte                ; if not, wait until it has

     bsf  CA1                              ; deassert acknowledge line
     goto CheckSend                        ; if so, wait for next one
```

Master control software

This is the main chunk of software that controls the master control unit, and so controls the entire system. Mainly what it does is handles the menu system by sending it through the serial port. It was not possible to use a modem, because two telephone lines would be needed for testing, but the software would need little modification to do this. For the purposes of demonstration, only one appliance can be controlled by the software because there is only one receiver module, but this could easily be extended to up to 255 appliances.

The software is split into four chunks, all of which I have written and are thus included here. The main program controls the menu system and contains most of the code. The interrupt service routines handle interrupts generated by the various devices in the master control unit, and are mostly separate for the main program. The exception code handles any exceptions triggered by the 68000 by displaying a flashing pattern on the LED port. It also contains the beginnings of an operating system, as the TRAP command is used to provide access to a number of software routines as if they were additional instructions. In this version, the only routines present are real time clock accessing routines, as well as a means of calling instructions that can only be executed in supervisor mode. In a proper operating system, this would be extended to access large numbers of routines, in a similar way to that of the INT instruction of the 8086 and the SWI instruction of the ARM. Finally, the definitions file defines names for a number of numeric constants, which improves the clarity of the assembly listing.

The full software listings are shown in Appendix B.

When the unit is first reset, the following menu in sent through the master control unit's serial port, and appears on a terminal connected to it:

```
Home Control System
Version 1.09 (9 March 1997)

1)      Configure system
2)      Control appliance now
3)      Set timed event
4)      List current timed events
5)      Clear timed event
6)      View current conditions
7)      Goodbye

Please enter a number :
```

This is the main menu, which contains the main control options. Of these, 4 and 5 do not do anything in the current version of the software, but they are included to demonstrate what should be present in a production system. 6 merely shows a display of the current time, and so this should be improved as well.

Pressing 1 leads to the Configure Menu, which currently only allows the current time to be set:

```
Home Control System

Version 1.09 (9 March 1997)


1)      Set system clock

2)      Return to main menu


Please enter a number :
```

Pressing 1 here gives a series of prompts to enter the time:

```
Home Control System

Version 1.09 (9 March 1997)


Set system clock
---------------

Year: 1997
Month: 3
Day of month: 9
Hours: 10
Minutes: 37
Seconds: 22
```

Pressing 2 at the main menu allows you to control a device immediately. As only one device is implemented is this system, only the immersion heater can be controlled.

```
Home Control System

Version 1.09 (9 March 1997)


Control device now
-----------------

1)      Central heating
2)      Outside lights
3)      Ground floor lights
4)      Ground floor appliances
5)      First floor lights
6)      First floor appliances
7)      Immersion heater
8)      Other electrical devices

9)      Return to main menu


Please enter a number :
```

If a number other than 7 is pressed, a message is displayed saying that there is only one device controllable. If 7 is pressed, the user is asked whether they want to turn the device on, off, or proportionally control it. In this example, they have pressed 3 for proportional control, and so are prompted for a value. At present, this value is sent directly to the device, but the software could be modified to accept it as a percentage or as a temperature, for example.

```
Device control:

1) Device off
2) Device on
3) Proportional control

Please enter a number : 3
Value to send to device: 50
```

Option 3 from the main menu allows a timed event to be set. At present, the software will only support one event to occur in the next 24 hours. It brings up the same menu as for control timed event, so that the device can be selected. Once it has been selected, the system asks for the time and setting for that device.

```
Home Control System

Version 1.09 (9 March 1997)


Set timed event
---------------


Hours: 18
Minutes: 00
Seconds: 00
Device control:

1) Device off
2) Device on
3) Proportional control

Please enter a number : 3
Value to send to device: 25
```

After this, the system displays the main menu again. Option 7 on the main menu would hang up a modem if one were connected, as it sends +++ to enter modem command mode, and then ATH to hang up the telephone line.

Evaluation

In its current form, the system does work, but to produce a system for mass production would still need a lot of work. In most areas, the hardware that has been built is sufficient for production, but the software needs considerable improvement. Most of the limitations that currently exist in the system exist because the software has been written to demonstrate the hardware, and show what it could do in a production system. Many weeks or months could be spent improving the software to bring the system up to a marketable product. In particular, much needed features include support for modems, which I was not able to include because it needed two telephone lines for testing, which were not available and would have generate large bills in the process. The system should also have password protection, so that only the owner of the house can access the system, and support for more than one timed event. The hardware is capable of setting approximately 2,000 timed events which can be set up to 99 years ahead, but there was no time to implement this in the software.

The microprocessor system itself seems to be quite reliable – it was left on overnight, and was still running in the morning. This is a necessity in a system that it intending to be left running for long periods. Also, if the system does crash while the householder is away, it may leave safety critical devices on, such as an oven or an electric fire. If it has severely crashed, it may not come back on line, and so the householder would not be able to turn these devices off. No evidence was found that this would happen, and if it did because of power failure the power on reset should clear any problems. A useful addition to the system would be a watchdog timer chip, which has to be accessed by the system every few seconds. If this is not done, the watchdog timer will decide the system has crashed and assert the master reset line.

The emulated EPROM was not perfectly reliable, and occasionally did not write a value properly. After investigation, it seems that the most likely cause was the 1.8m cable which connected the microprocessor board to the parallel port of the computer. As the signals travelling along this cable are quite fast, they start to deteriorate after a short distance. This would not matter in a production system, as the emulated EPROM would have to be replaced with real EPROMs or mask programmed ROMs to ensure the program stayed intact when power was removed.

The system occupies a large amount of board area, which would ideally be reduced in a production system. This could be done by using 16 bit wide RAM and ROM, and employing a processor that contains some on-chip peripherals, such as the Motorola 68300 series. The large board area which contains just wiring could be significantly reduced by building the system on a four or six layer PCB. As the ROM would be real ROM, there would be no need for the many tristates used in the EPROM emulator.

The data link worked reasonably well, although it was occasionally subject to interference. This is probably because the transmitters are driven by the output of a logic gate. Ideally, there should be some form of power amplifier to boost the signal before it is transmitted. The receiver could also be improved, because it currently has a fixed gain. If the transmitted signal varies in strength, the receiver picks up these variations and this may cause the logic output signal to be lost when the strength is low. This could be improved by adding an automatic gain control to more greatly amplify weak signals. This would have to be very carefully designed to ensure that it discriminated between noise and signal.

Occasionally the data link would get very noisy and lose the signal altogether. The output signal could be changed by moving different metal items above the breadboard. This seems to have been due to the RF signals being generated interfering with other parts of the circuit, possibly creating a feedback loop. The metal objects absorb the electromagnetic radiation produced, and so affect the circuit. This effect could probably have been eliminated by building the transmitters and receivers in separate metal boxes.

The data link was evaluated numerically by passing a square wave through in each direction, and determining the maximum frequency of square wave that could be resolved. On the 10 kHz carrier, the communications channel could handle frequencies from 0.1 to 80 Hz, while on the 100 kHz channel, it could handle from 0.1 to 2500 Hz. On both square wave outputs, there was a blurred rising edge of the square wave, and it was this that limited the bandwidth. The most likely cause of this is a delay in the transmission oscillator starting up, in which the output level is unstable. This gets transmitted through the various filters and amplifiers and is converted to a blur on the logic output. Careful filtering would be able to reduce this. As it is, the PIC does not appear to suffer a problem with this.

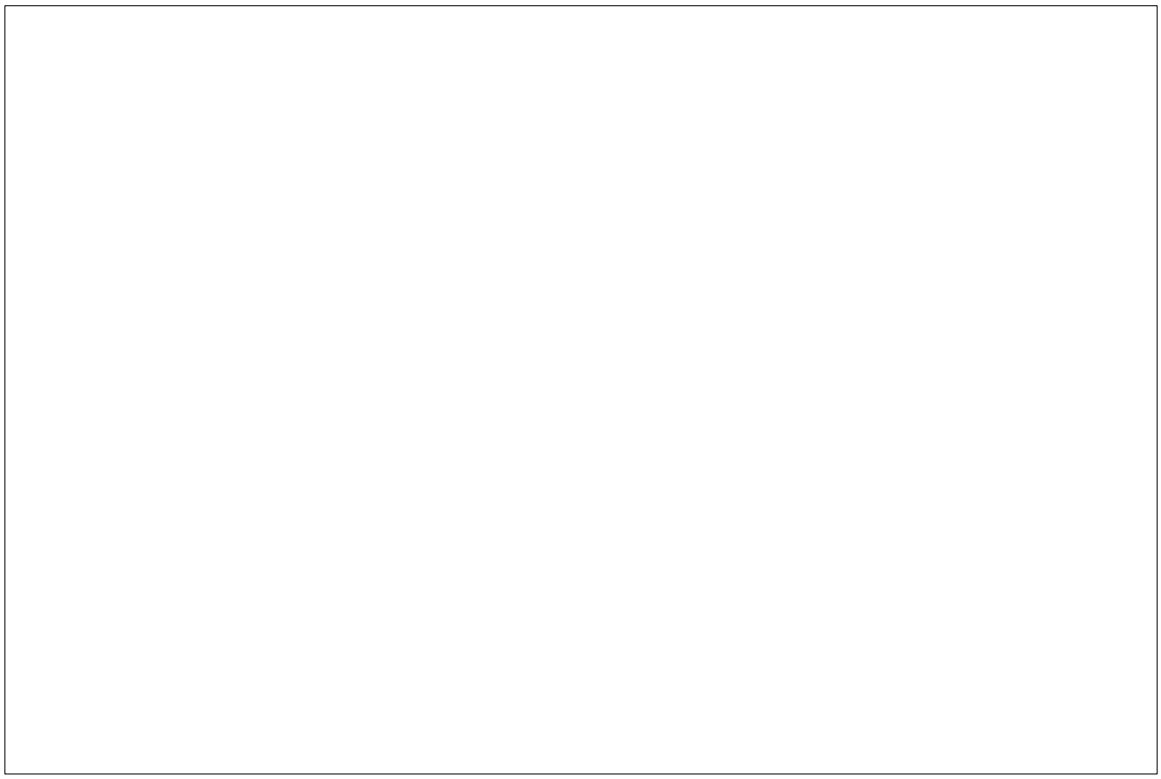
As stated previously, if this were a production version, the data link would use a direct mains connection to avoid the filtering effect of power transformers. If direct mains access were possible in this project, I would probably have investigated one of the mains carrier transceiver ICs designed for this purpose, such as the National Semiconductor LM1893. As it is, the system being used for the data link could be modified for mains use, and would probably work reasonably well if the improvements mentioned above were implemented. Mains is a much more noisy medium than a length of wire, so sharper filtering would have to be used.

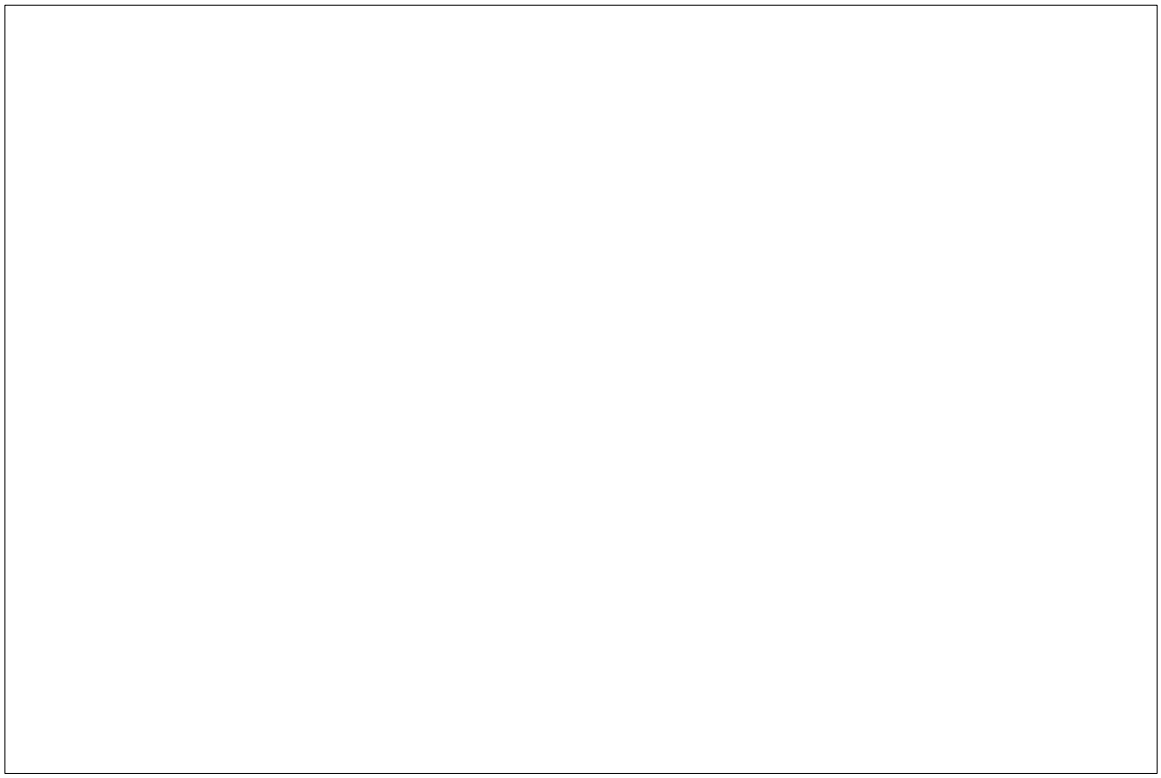
The PICs at each end of the data link seemed to be working reasonably well, although they were occasionally susceptible to noise introduced in the data link. This could be reduced or even eliminated from a digital point of view by adding error correction to their software. This system is similar to that used on CD players to ignore scratches, and involves sending more data than is required, so that the receiving system can detect if there is an error in it, and reconstruct the original data packet from the extra information.

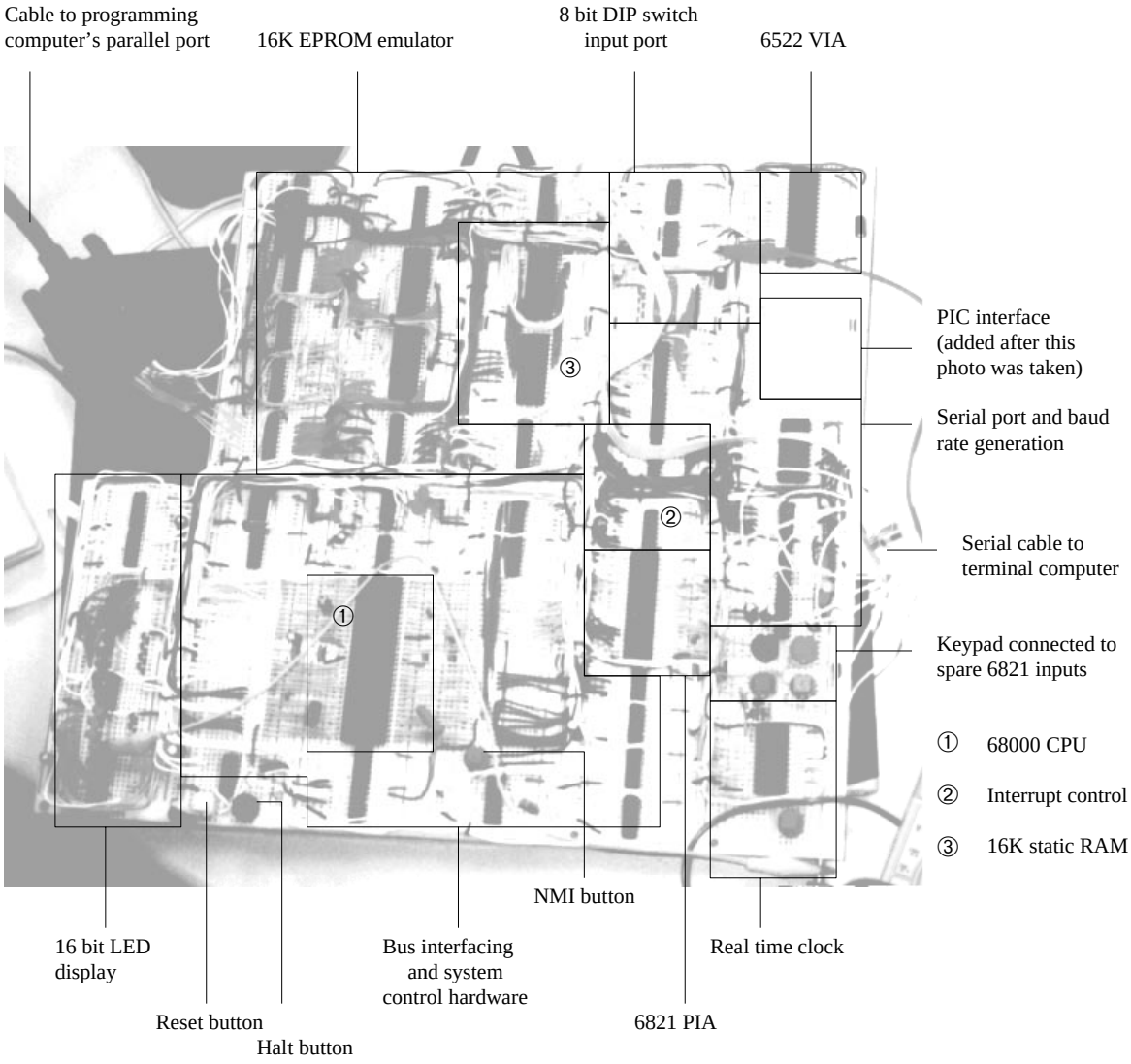
In summary, the system worked reasonably well for what it did, and showed that the hardware was working and sufficient for the task, but the software needs considerable refinement if this were to be a marketable product. The limitations of no direct mains access meant that the data link was different to that would be used in a production system, and this would need modification if this were to be produced commercially. A mains link is much more convenient, because it does not need extra wiring around the home.

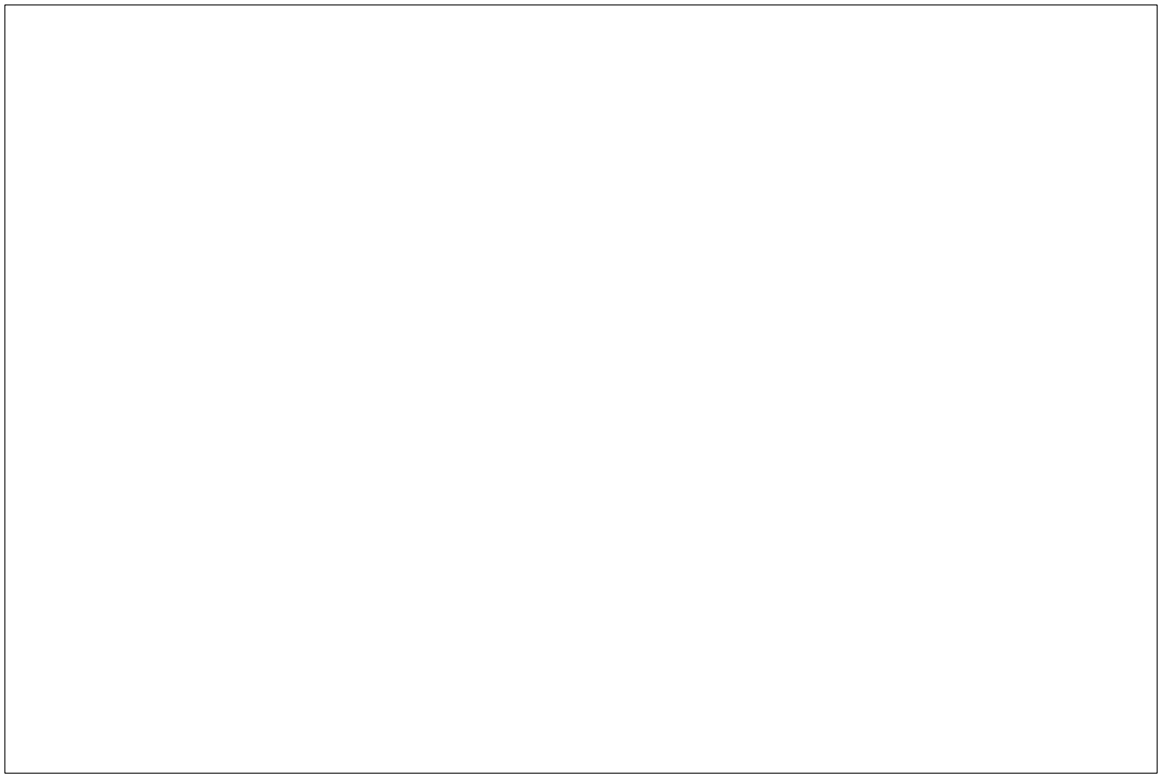
Photographs of project

The microprocessor system

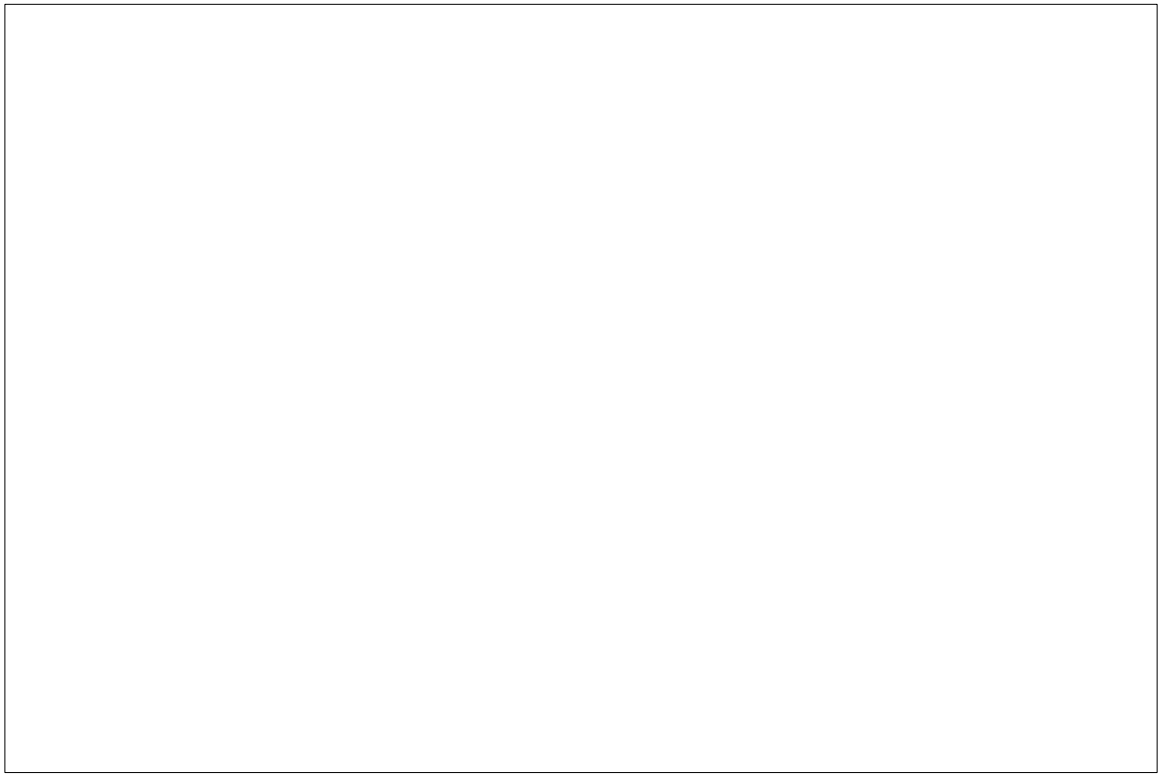




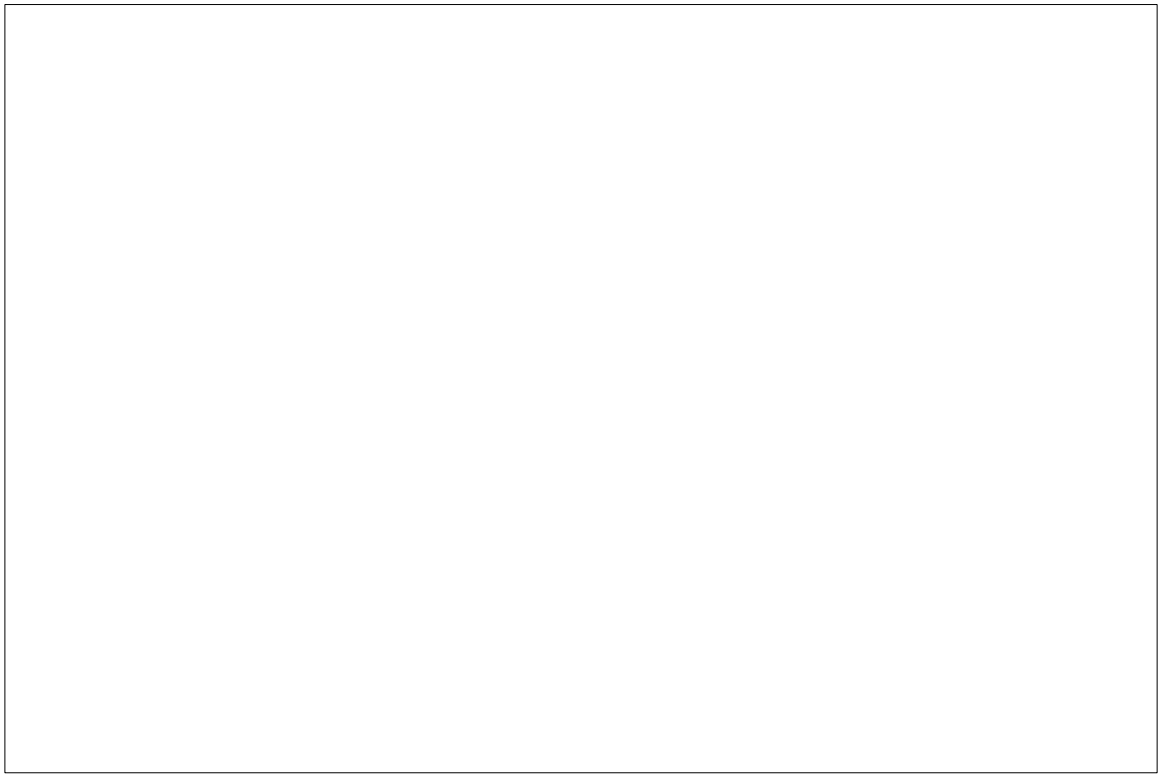




Data link and remote module



Non-functional mains data link circuit



Wire data link circuit (both ends) and remote module logic board

Appendix A

Excerpts from ‘M68000 Microprocessor User’s Manual’

(copyright Motorola Corporation)

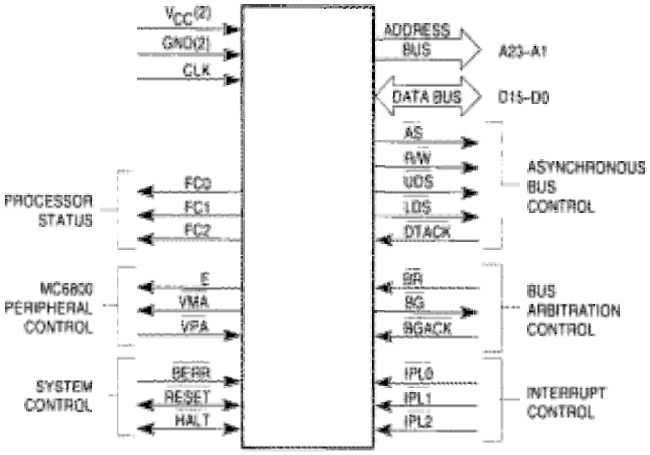


Figure 3-1. Input and Output Signals (MC68000, MC68HC000 and MC68010)

Table 3-3. Function Code Outputs

Function Code Output			
FC2	FC1	FC0	Address Space Type
Low	Low	Low	(Undefined, Reserved)
Low	Low	High	User Data
Low	High	Low	User Program
Low	High	High	(Undefined, Reserved)
High	Low	Low	(Undefined, Reserved)
High	Low	High	Supervisor Data
High	High	Low	Supervisor Program
High	High	High	CPU Space

Bitmap removed to save space
Copy available from http://www.mot.com

Table 6-2. Exception Vector Assignment

Vectors Numbers		Address			
Hex	Decimal	Dec	Hex	Space[6]	Assignment
0	0	0	000	SP	Reset: Initial SSP[2]
1	1	4	004	SP	Reset: Initial PC[2]
2	2	8	008	SD	Bus Error
3	3	12	00C	SD	Address Error
4	4	16	010	SD	Illegal Instruction
5	5	20	014	SD	Zero Divide
6	6	24	018	SD	CHK Instruction
7	7	28	01C	SD	TRAPV Instruction
8	8	32	020	SD	Privilege Violation
9	9	36	024	SD	Trace
A	10	40	028	SD	Line 1010 Emulator
B	11	44	02C	SD	Line 1111 Emulator
C	12[1]	48	030	SD	(Unassigned, Reserved)
D	13[1]	52	034	SD	(Unassigned, Reserved)
E	14	56	038	SD	Format Error[5]
F	15	60	03C	SD	Uninitialized Interrupt Vector
10–17	16–23[1]	64	040	SD	(Unassigned, Reserved)
		92	05C		
18	24	96	060	SD	Spurious Interrupt[3]
19	25	100	064	SD	Level 1 Interrupt Autovector
1A	26	104	068	SD	Level 2 Interrupt Autovector
1B	27	108	06C	SD	Level 3 Interrupt Autovector
1C	28	112	070	SD	Level 4 Interrupt Autovector
1D	29	116	074	SD	Level 5 Interrupt Autovector
1E	30	120	078	SD	Level 6 Interrupt Autovector
1F	31	124	07C	SD	Level 7 Interrupt Autovector
20–2F	32–47	128	080	SD	TRAP Instruction Vectors[4]
		188	0BC		—
30–3F	48–63[1]	192	0C0	SD	(Unassigned, Reserved)
		255	0FF		
40–FF	64–255	256	100	SD	User Interrupt Vectors
		1020	3FC		—

NOTES:

1. Vector numbers 12, 13, 16–23, and 48–63 are reserved for future enhancements by Motorola. No user peripheral devices should be assigned these numbers.
2. Reset vector (0) requires four words, unlike the other vectors which only require two words, and is located in the supervisor program space.
3. The spurious interrupt vector is taken when there is a bus error indication during interrupt processing.
4. TRAP #n uses vector number 32+ n.
5. MC68010 only. This vector is unassigned, reserved on the MC68000 and MC68008.
6. SP denotes supervisor program space, and SD denotes supervisor data space.

Appendix B

Main program

```
        INCLUDE x68.defs

trap_code       EQU     $200
interrupt_code  EQU     $400

        INCLUDE x68.excs
        INCLUDE x68.intserv4

        ORG     $800                    * base of ROM after exception table
start
        BRA     main_init               * main initialisation
start_init
        BSR     serial_init             * initialise serial system
        BSR     pia_init                * initialise PIA
        BSR     rtc_init                * initialise clock
        BSR     via_init                * set VIA to talk to PIC
        BSR     program_init




main_menu
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * for banner, and send
        LEA     mmstrtbl,A1             * point to table of string pointers
        BSR     send_stringtable        * for main menu, and send

        MOVEQ   #'7',D1                 * maximum number to accept
        BSR     num_wait                * wait for a character
        MOVE.L  D0,D2
        MOVE.W  #$8000,D1               * provide a small delay so that the
main_menu_char_delay                    * character can be seen on terminal
        DBF     D1,main_menu_char_delay
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear screen
        SUB.B   #'1',D2                 * convert ASCII number into
        CMP.B   #0,D2                   * binary
        BEQ     system_config
        CMP.B   #1,D2
        BEQ     control_now
        CMP.B   #2,D2
        BEQ     set_event
        CMP.B   #5,D2
        BEQ     rtc_showtime
        CMP.B   #6,D2
        BEQ     quit
        BRA     main_menu

quit
        LEA     modem_cmdmode,A0        * pointer to modem wakeup string
        BSR     send_string
        MOVEQ   #5,D1
```

```
quit_loop1
        MOVE.L  #$FFFF,D0               * Effectively a delay loop value
quit_loop2                              * of $5FFFF
        DBF     D0,quit_loop2           * wait for a seconds until modem
        DBF     D1,quit_loop1           * responds
        LEA.l   modem_hangup,A0
        BSR     send_string             * send hangup command
quit_waitfornodcd
        MOVE.B  ACIA_SR,D0              * get status register
        BTST    #2,D0                   * is DCD bit high
        BEQ     quit_waitfornodcd       * if not wait until it is
        CLR     D0
quit_endloop
        ADDQ    #1,D0
        BRA     quit_endloop

system_config
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * for banner, and send
        LEA     configstrtbl,A1         * point to table of string pointers
        BSR     send_stringtable        * for config menu, and send
        MOVE.B  #'2',D1                 * maximum number to accept
        BSR     num_wait                * get key character
        MOVE.L  D0,D2
        MOVE.W  #$8000,D1               * provide a small delay so that the
system_config_char_delay                * character can be seen on terminal
        DBF     D1,system_config_char_delay
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear screen
        SUB.B   #'1',D2                 * convert ASCII number into binary
        CMP.B   #0,D2                   * if 1 pressed, set time
        BEQ     rtc_settime
        CMP.B   #1,D2                   * if 2 pressed, go to main menu
        BEQ     main_menu
        BRA     system_config


num_wait                                * wait for a number to be entered
                                        * must be 1 digit, between 1 and D1
        BSR     serial_getc             * get character
        BEQ     num_wait        * if no char ready, wait for one
        CMP.B   #'1',D0                 * is it a number 1-2
        BLT     num_wait        * if not wait for a number
        CMP.B   D1,D0
        BGT     num_wait
        BSR     serial_putc             * send the character
        RTS

rtc_settime
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * print banner
        LEA     clksetstrtbl,A1
        BSR     send_stringtable        * print title

        LEA     rtcsettbl,A3            * table pointing to prompt and regs
rtc_settime_loop
```

```
        MOVEQ   #CR,D0
        BSR     serial_putc
        MOVEQ   #LF,D0
        BSR     serial_putc
        MOVEA.L (A3)+,A2                * get prompt ptr
        MOVE.W  (A3)+,D3                * get min value
        MOVE.W  (A3)+,D4                * get max value
        BSR     get_bcd_number
        MOVE.W  (A3)+,D1                * RTC register
        MOVE.L  D0,D2                   * value to store
        MOVE.W  #Hardware_RTC_Write,D0  * call OS routine
        TRAP    #15
        MOVE.L  (A3),D0                 * get next address
        TST     D0                      * is it zero?
        BNE     rtc_settime_loop        * if not, get next value
        BRA     system_config

get_bcd_number
        MOVEA.L A2,A0
        BSR     send_string             * print prompt
        BSR     get_line                * read line
        BSR.S   bcdval_line             * extract BCD data from line
        CMP.W   D3,D0
        BLT.S   get_bcd_number_again
        CMP.W   D4,D0                   * is it past max value
        BGT.S   get_bcd_number_again
        RTS

get_bcd_number_again
        LEA     cursorup,A0
        BSR     send_string
        LEA     wipestr,A0
        BSR     send_string
        BRA     get_bcd_number

bcdval_line                             * convert string pointed to by A0
        MOVEM.L D1/A0,-(A7)             * to BCD number
        MOVE.B  (A0)+,D1                * get character
        CMP.B   #‘0’,D1                 * check it is a valid digit
        BLT.S   bcdval_line_err
        CMP.B   #‘9’,D1
        BGT.S   bcdval_line_err
        SUB.B   #‘0’,D1                 * convert to binary digit
        LSL.W   #4,D1                   * make into MSB
        MOVE.W  D1,D0                   * put in D0
        MOVE.B  (A0)+,D1                * get character
        CMP.B   #32,D1
        BLT.S   bcdval_line_singdig
        CMP.B   #‘0’,D1                 * check it is a valid digit
        BLT.S   bcdval_line_err
        CMP.B   #‘9’,D1
        BGT.S   bcdval_line_err
        SUB.B   #‘0’,D1                 * convert to binary digit
        OR.W    D1,D0                   * put in LSB of D0
        AND.W   #$FF,D0
        MOVEM.L (A7)+,D1/A0
        RTS                             * return value in D0

bcdval_line_err
        MOVEQ   #-1,D0                  * return invalid result
```

```
        MOVEM.L (A7)+,D1/A0
        RTS

bcdval_line_singdig                     * if second digit is ctrl char,
        LSR.W   #4,D0                   * treat as 1 digit
        AND.W   #$0F,D0
        MOVEM.L (A7)+,D1/A0
        RTS

control_now
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * for banner, and send
        LEA     connowstrtbl,A1         * point to table of string pointers
        BSR     send_stringtable        * for config menu, and send
        MOVE.B  #'9',D1                 * maximum number to accept
        BSR     num_wait                * get key character
        MOVE.L  D0,D2
        MOVE.W  #$8000,D1               * provide a small delay so that the
control_now_char_delay                  * character can be seen on terminal
        DBF     D1,control_now_char_delay
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear screen
        SUB.B   #'1',D2                 * convert ASCII number into binary
        CMP.B   #8,D2                   * if 9 pressed, go to main menu
        BEQ     main_menu
        CMP.B   #7,D2                   * if 8 pressed, get number of other
        BEQ     control_now_other_device * device and control it
        CMP.B   #6,D2                   * if 7 pressed, get number of heater
        BEQ     control_now_heater_device * and control it
        BRA     not_implemented         * if other number, say isn't
                                        * implemented

control_now_other_device
        BSR     other_device
        BRA     control_device

control_now_heater_device
        BSR     heater_device
        BRA     control_device


not_implemented
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * for banner, and send
        LEA     notimpstrtbl,A1         * point to table of string pointers
        BSR     send_stringtable        * for message, and send
not_implemented_wait
        BSR     serial_getc
        BEQ     not_implemented_wait    * wait for key to be pressed
        BRA     main_menu

other_device
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * for banner, and send
```

```
        CLR     D3                      * minimum number
        MOVE.W  #$99,D4                 * maximum number
        LEA     od1,A2                  * get prompt
        BSR     get_bcd_number          * get the number - should be 0-255,
                                        * but is simpler if get_bcd is used
                                        * therefore only 0-99 available
        BSR     hex_to_dec              * convert to hex number in D0
        RTS                             * return

heater_device
        MOVE.W  heater_dev_no,D0* read heater device number
        RTS

control_device
        MOVE.L  D0,D6                   * keep device number safe
        MOVEQ   #$67,D7                 * code for control now
        BRA     set_event_get_setting   * find out what to set to

control_device_now                      * D6=device number, D0=setting
        MOVE.L  D6,D1                   * put number in D1
        EXG.L   D0,D1                   * swap number and setting
        BSR     send_PIC_command        * send command packet
        BRA     main_menu               * then go to main menu

set_event
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * for banner, and send
        LEA     setevstrtbl,A1          * point to table of string pointers
        BSR     send_stringtable        * for config menu, and send
        MOVE.B  #'9',D1                 * maximum number to accept
        BSR     num_wait                * get key character
        MOVE.L  D0,D2
        MOVE.W  #$8000,D1               * provide a small delay so that the
set_event_char_delay                    * character can be seen on terminal
        DBF     D1,set_event_char_delay
        MOVEQ   #FF,D0
        BSR     serial_putc             * clear screen
        SUB.B   #'1',D2                 * convert ASCII number into binary
        CMP.B   #8,D2                   * if 9 pressed, go to main menu
        BEQ     main_menu
        CMP.B   #7,D2                   * if 8 pressed, get number of other
        BEQ     set_event_other_device  * device and control it
        CMP.B   #6,D2                   * if 7 pressed, get number of heater
        BEQ     set_event_heater_device * and control it
        BRA     not_implemented         * if other number, say isn't
                                        * implemented

set_event_other_device
        BSR     other_device
        BRA     set_event_get_details

set_event_heater_device
        BSR     heater_device
        BRA     set_event_get_details

set_event_get_details

        MOVE.L  D0,D2
```

```
        MOVEQ   #RTC_DEVICE_ALRM,D1     * device number byte in CMOS
        MOVE.W  #Hardware_RTC_Write,D0  * write it
        TRAP    #15

        MOVEQ   #FF,D0
        BSR     serial_putc             * clear the terminal screen
        LEA     banstrtbl,A1            * point to table of string pointers
        BSR     send_stringtable        * print banner
        LEA     setdetstrtbl,A1
        BSR     send_stringtable        * print title

        LEA     setdettbl,A3            * table pointing to prompt and regs
set_event_get_details_loop

        MOVEQ   #CR,D0
        BSR     serial_putc
        MOVEQ   #LF,D0
        BSR     serial_putc
        MOVEA.L (A3)+,A2                * get prompt ptr
        MOVE.W  (A3)+,D3                * get min value
        MOVE.W  (A3)+,D4                * get max value
        BSR     get_bcd_number
        MOVE.W  (A3)+,D1                * RTC register
        MOVE.L  D0,D2                   * value to store
        MOVE.W  #Hardware_RTC_Write,D0  * call OS routine
        TRAP    #15
        MOVE.L  (A3),D0                 * get next address
        TST     D0                      * is it zero?
        BNE     set_event_get_details_loop      * if not, get next value

set_event_get_setting
        LEA     newline,A0
        BSR     send_string             * print newline
        LEA     devctrlstrtbl,A1
        BSR     send_stringtable        * print menu
        MOVE.B  #'3',D1                 * maximum number to accept
        BSR     num_wait                * get key character
        MOVE.L  D0,D2
        MOVE.W  #$8000,D1               * provide a small delay so that the
set_event_get_details_char_delay        * character can be seen on terminal
        DBF     D1,set_event_get_details_char_delay
        SUB.B   #'1',D2                 * convert ASCII number into binary
        CMP.B   #0,D2                   * if 1 pressed, turn off
        BEQ     set_event_gd_off
        CMP.B   #1,D2                   * if 8 pressed, turn on
        BEQ     set_event_gd_on
                                        * here 2 was pressed, so get number
        LEA     newline,A1
        MOVEA.L A1,A0
        BSR     send_string

        LEA     value,A2
        MOVEQ   #0,D3
        MOVE.W  #$99,D4
        BSR     get_bcd_number
        CMP     #$67,D7                 * control immediately?
        BEQ     control_device_now
set_event_store_data
        MOVEQ   #RTC_DATA_VALUE,D1
        MOVE.L  D0,D2
```

```
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15

set_event_enable_irq
        MOVEQ   #RTC_STATUS_B,D1        * RTC register B
        MOVE.W  #Hardware_RTC_Read,D0   * read it
        TRAP    #15
        BSET    #5,D0                   * set alarm interrupt enable
        MOVE.L  D0,D2
        MOVEQ   #RTC_STATUS_B,D1        * RTC register B
        MOVE.W  #Hardware_RTC_Write,D0  * write it
        TRAP    #15
        BRA     main_menu

set_event_gd_off
        MOVEQ   #0,D0
        BRA     set_event_store_data

set_event_gd_on
        MOVE.W  #$99,D0
        BRA     set_event_store_data

send_PIC_command                        * D0 = device number, D1 = value
        BSR     send_PIC_byte           * send D0
        MOVE.L  D1,D0
        BSR     send_PIC_byte           * send D1
        MOVE.W  #$FF,D0
        BSR     send_PIC_byte           * send end of packet code
        RTS

send_PIC_byte
        MOVE.B  D0,VIA_DRA              * write value to PIC through VIA
        MOVE.W  #$FFF,D7                * delay value
send_PIC_byte_loop                      * delay waiting for IRQ
        DBF     D7,send_PIC_byte_loop
        RTS


* Initialisation section for main program

main_init
        MOVE.W  #$0,SR                  * clear status register - go into
                                        * user mode

        MOVEA.L #$43F00,A7              * set user stack pointer to
                                        * sensible RAM address
        BRA      start_init             * return - can't use BSR/RTS because
                                        * stacks have been swapped

pia_init
        MOVE.B  #$31,PIA_CRA            * access DDRA, enable CA1
                                        * and system interrupts
        MOVE.B  #$FF,PIA_DDRA           * make PIA Port A outputs
        MOVE.B  #$00,PIA_CRB            * set PB0-3 outputs,
        MOVE.B  #$0F,PIA_DDRB           *     PB4-7 inputs
        MOVE.B  #$04,PIA_CRB            * select Peripheral Reg B
        MOVE.B  #$F9,PIA_PRB            * other bits high, R-/W high,
                                        * DS, AS low, /CS high
        RTS                             * return
```

```
serial_init
        MOVE.B  DIN,D0                  * read input port
        AND.B   #$7F,D0                 * mask off other bits
        MOVE.B  D0,D1                   * take copy of D0
        AND.B   #$7,D0                  * mask only bps selector bits
        MOVE.B  D0,SER_BPS              * set serial data rate

        MOVE.B  #3,ACIA_CR              * Initialise 6850

        MOVEQ   #2,D2                   * select divide by 64

        BTST    #3,D1                   * is bit 3 of DIP switch on?
        BNE.S   serial_init_setdiv16    * if so select div by 16
serial_init_cont1
        AND.B   #$70,D1                 * get word select bits
        LSR.B   #2,D1                   * shift into correct position for
                                        * ACIA control register
        OR.B    D1,D2                   * OR with D2 -> D2
        *OR.B   #$00,D2                 * CR7-5 = 101, RTS on, enable
                                        * receive and transmit IRQs
        MOVE.B  D2,ACIA_CR              * set control register
        MOVE.B  D2,SERIAL_CONFIG
        CLR.W   SERIAL_RX_INPTR         * Reset buffer pointers
        CLR.W   SERIAL_RX_OUTPTR
        CLR.W   SERIAL_TX_INPTR
        CLR.W   SERIAL_TX_OUTPTR
        RTS                             * return from subroutine

serial_init_setdiv16
        MOVEQ   #1,D2                   * select divide by 16
        BRA     serial_init_cont1

rtc_init
        MOVEQ   #RTC_STATUS_D,D1        * RTC status register D
        MOVE.W  #Hardware_RTC_Read,D0   * read it
        TRAP    #15
        MOVE.L  D0,D3                   * Preserve result

        MOVEQ   #RTC_STATUS_A,D1        * RTC status register A
        MOVEQ   #$00,D2                 * clear it
        MOVE.W  #Hardware_RTC_Write,D0  * call OS routine
        TRAP    #15
        MOVEQ   #RTC_STATUS_C,D1        * RTC status register C
        MOVE.W  #Hardware_RTC_Read,D0   * reset interrupt flags
        TRAP    #15                     * call OS routine
        MOVEQ   #RTC_STATUS_B,D1        * RTC status register B
        MOVEQ   #$10,D2                 * set Update Interrupt Enabled
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15

        BTST    #7,D3                   * is bit 7 of reg D (VRT) set?
        BNE     rtc_init_end            * if so, don't bother to reset time

        MOVEQ   #RTC_SECS,D1            * on startup, set time and date
        MOVEQ   #$00,D2                 * to 00:00:00 on 1st January 1997
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15
        MOVEQ   #RTC_MINS,D1
        MOVEQ   #$00,D2
        MOVE.W  #Hardware_RTC_Write,D0
```

```
        TRAP    #15
        MOVEQ   #RTC_HRS,D1
        MOVEQ   #$00,D2
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15
        MOVEQ   #RTC_DAY,D1
        MOVEQ   #$1,D2
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15
        MOVEQ   #RTC_DATE,D1
        MOVEQ   #$1,D2
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15
        MOVEQ   #RTC_MONTH,D1
        MOVEQ   #$1,D2
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15
        MOVEQ   #RTC_YEAR,D1
        MOVE.b  #$97,D2
        MOVE.W  #Hardware_RTC_Write,D0
        TRAP    #15

rtc_init_end
        RTS

via_init                                * VIA initialisation
        CLR.B   VIA_DDRA                * Make port A an input
        CLR.B   VIA_DDRB                * Make port B an input
        MOVE.B  #%00001000,VIA_PCR      * Select port A handshake mode
        MOVE.B  #%10010000,VIA_IER      * Enable CB1 interrupt
        RTS                             * Return

program_init
        MOVE.L  #TF_BLK,TF_START        * initialise timeframe list
        MOVE.L  #TF_BLK,TF_END
        RTS

* Miscellaneous general purpose subroutines

serial_putc
        BTST    #1,ACIA_SR              * is port 1 ready for a character?
        BEQ     serial_putc             * if not, wait for it
        MOVE.B  D0,ACIA_TR              * out it goes.
        RTS

serial_getc
        BTST    #0,ACIA_SR              * is character ready?
        BEQ.S   serial_getc_ret         * if not, return Zero status
        MOVE.B  ACIA_RR,D0              * else get the character
        AND.B   #$7F,D0                 * zero out the high bit
serial_getc_ret
        RTS


send_string                             * Send serial string pointed to by A0
        MOVE.B  (A0)+,D0                * get next byte
        TST.B   D0
        BEQ     send_string_return
        BSR     serial_putc             * send it
        BRA     send_string             * if not do next one
```

```
send_string_return
        RTS

send_bcd_number
        MOVE.L  D1,-(A7)
        MOVE.L  D0,D1
        LSR     #4,D0
        AND     #$0F,D0
        ADD.B   #'0',D0
        BSR     serial_putc
        MOVE.L  D1,D0
        AND     #$0F,D0
        ADD.B   #'0',D0
        BSR     serial_putc
        MOVE.L  (A7)+,D1
        RTS

send_stringtable                        * send an whole string table through
                                        * the serial port
                                        * on entry A1 points to table
        MOVEM.L D1/A0,-(A7)
        CLR     D1
send_stringtable_loop
        MOVEA.L (A1,D1),A0              * get address from table
        CMPA    #0,A0
        BEQ     send_stringtable_end    * if zero, end procedure
        BSR     send_string             * if not, send string
        ADDQ    #4,D1
        BRA     send_stringtable_loop
send_stringtable_end
        MOVEM.L (A7)+,D1/A0             * restore registers
        RTS                             * return

get_line                                * read a line, terminated by control
                                        * character from serial port
        LEA     SERIAL_WORKSPACE,A0     * buffer to store string
get_line_loop
        BSR     serial_getc             * get character
        BEQ     get_line_loop
        CMP.B   #127,D0                 * is it delete/backspace
        BEQ     get_line_del            * if so delete character
        CMP.B   #8,D0
        BEQ     get_line_del
        BSR     serial_putc             * otherwise echo it to the terminal
        MOVE.B  D0,(A0)+
        CMP.B   #31,D0                  * is it control char?
        BGT     get_line_loop           * if not, get next
        MOVE.B  #0,-1(A0)               * replace control char with zero
        LEA     SERIAL_WORKSPACE,A0     * return start of string
        RTS

get_line_del
        SUBQ    #1,A0                   * delete previous char
        CMPA.L  #SERIAL_WORKSPACE,A0    * is it before workspace
        BLT.S   get_line                * if so, start again
        MOVEQ   #8,D0                   * backspace
        BSR     serial_putc
        MOVEQ   #' ',D0                 * overwrite last char
        BSR     serial_putc
        MOVEQ   #8,D0                   * backspace
```

```
        BSR     serial_putc             * delete character on terminal
        BRA.S   get_line_loop           * and delete char

rtc_sendtime
        LEA     tsec,A0
        BSR     send_string
        MOVE.B  RTC_SC_SECS,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        LEA     tmin,A0
        BSR     send_string
        MOVE.B  RTC_SC_MINS,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        LEA     thrs,A0
        BSR     send_string
        MOVE.B  RTC_SC_HRS,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        LEA     tday,A0
        BSR     send_string
        MOVE.B  RTC_SC_DAY,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        LEA     tdate,A0
        BSR     send_string
        MOVE.B  RTC_SC_DATE,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        LEA     tmonth,A0
        BSR     send_string
        MOVE.B  RTC_SC_MONTH,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        LEA     tyear,A0
        BSR     send_string
        MOVE.B  RTC_SC_YEAR,D0
        BSR     send_bcd_number
        LEA     newline,A0
        BSR     send_string
        RTS
rtc_showtime
        MOVEQ   #FF,D0
        BSR     serial_putc
rtc_showloop
        BSR     rtc_sendtime
        MOVE.W  #$1000,D0
rl
        DBF     D0,rl
        lea     up,A0
        BSR     send_string
        BSR     serial_getc
        BNE     main_menu
        BRA     rtc_showloop
```

```
        DC.L 0

hex_to_dec
        MOVE.W  D0,D1                   * D0 = BCD value
                                        * D1 = decimal result
        AND.W   #$F,D1                  * mask off high bits
        AND.W   #$F0,D0                 * mask off low bits of D0
        LSR.W   #4,D0                   * make into nybble
        MOVE.L  D0,D2                   * D2 is temp reg
        LSL.W   #3,D2                   * D0 * 8
        ADD.W   D0,D2                   * D0 * 8 + D0
        ADD.W   D0,D2                   * D0 * 8 + D0 + D0 = 10 D0
        ADD.W   D1,D2                   * add to low nybble and ret in D0
        MOVE.W  D2,D0
        RTS

* Data and text messages

heater_dev_no
        DC.W    $0007                   * PIC id number for heater
        DCB.W   0,0
tb
        DC.B    'Home Control System',LF,CR,LF,CR,0
        DCB.W   0,0

ver
        DC.B    'Version 1.10 (9 March 1997)',LF,CR,LF,CR,LF,CR,0
        DCB.W   0,0

* Main menu options
mm
mm1     DC.B    '1)     Configure system',LF,CR,LF,CR,0
mm2     DC.B    '2)     Control appliance now',LF,CR,LF,CR,0
mm3     DC.B    '3)     Set timed event',LF,CR,LF,CR,0
mm4     DC.B    '4)     List current timed events',LF,CR,LF,CR,0
mm5     DC.B    '5)     Clear timed event',LF,CR,LF,CR,0
mm6     DC.B    '6)     View current conditions',LF,CR,LF,CR,0
mm7     DC.B    '7)     Goodbye',LF,CR,LF,CR,0
gap     DC.B    LF,CR,0
plent   DC.B    'Please enter a number : ',0
        DC.W    0

banstrtbl
        DC.L    tb
        DC.L    ver
        DC.L    0
mmstrtbl
        DC.L    mm1
        DC.L    mm2
        DC.L    mm3
        DC.L    mm4
        DC.L    mm5
        DC.L    mm6
        DC.L    mm7
        DC.L    gap
        DC.L    plent
        DC.L    0

cs
cs1     DC.B    '1)     Set system clock',LF,CR,LF,CR,0
```

```
cs2     DC.B    ‘2)     Return to main menu’,LF,CR,LF,CR,0
        DC.L    0

configstrtbl
        DC.L    cs1
        DC.L    cs2
        DC.L    gap
        DC.L    plent
        DC.L    0

sc
sc1     DC.B    ‘Set system clock’,LF,CR,0
sc2     DC.B    ‘————————’,LF,CR,0
        DC.W    0

clksetstrtbl
        DC.L    sc1
        DC.L    sc2
        DC.L    0

modem_cmdmode
        DC.B    ‘+++’,0
modem_hangup
        DC.B    ‘ATH’,CR,0
        dc.l    0

tsec    DC.B    ‘Seconds: ’,0
tmin    DC.B    ‘Minutes: ’,0
thrs    DC.B    ‘Hours: ’,0
tday    DC.B    ‘Day of week: ’,0
tdate   DC.B    ‘Day of month: ’,0
tmonth  DC.B    ‘Month: ’,0
tyear   DC.B    ‘Year: 19’,0
        DCB.W   0,0

rtcsettbl
        DC.L    tyear
        DC.W    $00,$99,RTC_YEAR
        DC.L    tmonth
        DC.W    $01,$12,RTC_MONTH
        DC.L    tdate
        DC.W    $01,$31,RTC_DATE
        DC.L    thrs
        DC.W    $00,$23,RTC_HRS
        DC.L    tmin
        DC.W    $00,$59,RTC_MINS
        DC.L    tsec
        DC.W    $00,$59,RTC_SECS
        DC.L    0
        DC.W    0,0

cn      DC.B    ‘Control device now’,LF,CR,0
cns     DC.B    ‘——————————’,LF,CR,LF,CR,0
ct1     DC.B    ‘1)     Central heating’,LF,CR,0
ct2     DC.B    ‘2)     Outside lights’,LF,CR,0
ct3     DC.B    ‘3)     Ground floor lights’,LF,CR,0
ct4     DC.B    ‘4)     Ground floor appliances’,LF,CR,0
ct5     DC.B    ‘5)     First floor lights’,LF,CR,0
ct6     DC.B    ‘6)     First floor appliances’,LF,CR,0
ct7     DC.B    ‘7)     Immersion heater’,LF,CR,0
```

```
ct8     DC.B    ‘8)     Other electrical devices’,LF,CR,LF,CR,0
ct9     DC.B    ‘9)     Return to main menu’,LF,CR,LF,CR,0
        DCB.W   0,0

connowstrtbl
        DC.L    cn
        DC.L    cns
        DC.L    ct1
        DC.L    ct2
        DC.L    ct3
        DC.L    ct4
        DC.L    ct5
        DC.L    ct6
        DC.L    ct7
        DC.L    ct8
        DC.L    ct9
        DC.L    gap
        DC.L    plent
        DC.L    0

se      DC.B    ‘Set timed event’,LF,CR,0
ses     DC.B    ‘————————-’,LF,CR,LF,CR,0
        DCB.W   0,0

setevstrtbl
        DC.L    se
        DC.L    ses
        DC.L    ct1
        DC.L    ct2
        DC.L    ct3
        DC.L    ct4
        DC.L    ct5
        DC.L    ct6
        DC.L    ct7
        DC.L    ct8
        DC.L    ct9
        DC.L    gap
        DC.L    plent
        DC.L    0

        DCB.W   0,0
ni1     DC.B    ‘This function is not implemented in this version’,LF,CR,0
ni2     DC.B    ‘of the control software. Currently, only the’,LF,CR,0
ni3     DC.B    ‘immersion heater can be controlled.’,LF,CR,LF,CR,0
ni4     DC.B    ‘If this system were being used commercially,’,LF,CR,0
ni5     DC.B    ‘these functions would be present and the software’,LF,CR,0
ni6     DC.B    ‘would be considerably refined.’,LF,CR,LF,CR,0
nip     DC.B    ‘Press any key to continue.’,LF,CR,0

        DCB.W   0,0
notimpstrtbl
        DC.L    ni1
        DC.L    ni2
        DC.L    ni3
        DC.L    ni4
        DC.L    ni5
        DC.L    ni6
        DC.L    nip
        DC.L    0
```

```
        DCB.W   0,0
od1     DC.B    'Enter number of device to control: ',0

setdetstrtbl
        DC.L    se
        DC.L    ses
        DC.L    0

setdettbl
        DC.L    thrs
        DC.W    $00,$23,RTC_HRS_ALRM
        DC.L    tmin
        DC.W    $00,$59,RTC_MINS_ALRM
        DC.L    tsec
        DC.W    $00,$59,RTC_SECS_ALRM
        *DC.L   value
        *DC.W   $00,$99,$0F
        DC.L    0
        DC.W    0,0

dc
dc1     DC.B    'Device control:',LF,CR,LF,CR,0
dc2     DC.B    '1) Device off',LF,CR,0
dc3     DC.B    '2) Device on',LF,CR,0
dc4     DC.B    '3) Proportional control',LF,CR,0
        DCB.W   0,0

devctrlstrtbl
        DC.L    dc1
        DC.L    dc2
        DC.L    dc3
        DC.L    dc4
        DC.L    gap
        DC.L    plent
        DC.L    0

proptbl
        DC.L    value
        DC.W    $00,$99,$0F
        DC.L    0,0

value   DC.B    'Value to send to device: ',0
        DCB.W   0,0

newline DC.B    CR,LF,0

up      DC.B    27,'[1A',27,'[1A',27,'[1A',27,'[1A',27,'[1A',27,'[1A',27,'[1A',0

wipestr DCB.B   79,' '
        DCB.B   79,8
        DC.B    0

cursorup
        DC.B    27,'[1A',LF,0
        END start
```

Interrupt handling code

```
* Interrupt service routines

        ORG     $70                     * PIA's interrupt vector
        DC.L    pia_int

        ORG     $74                     * VIA's interrupt vector
        DC.L    via_int

        ORG     $7C                     * NMI handler
        DC.L    memdump

        ORG     interrupt_code          * Centrally allocated code space

int_return                              * general purpose return
        MOVEM.L (A7)+,ALLREGS           * restore regs
        RTE                             * return

pia_int                                 * Called on PIA interrupt
        MOVEM.L ALLREGS,-(A7)           * Preserve all registers on stack
        MOVE.B  PIA_CRA,D0              * Read PIA status register A

        BTST    #7,D0                   * Is IRQ A set
        BNE     rtc_int                 * its an RTC interrupt
        BTST    #6,D0                   * Is IRQ B set
        NOP                             * its something else's interrupt
        MOVE.B  PIA_CRB,D0              * Read PIA status register A
        BTST    #7,D0                   * Is IRQ A set
        NOP                             * its something else's interrupt
        BTST    #6,D0                   * Is IRQ B set
        NOP                             * its something else's interrupt
        MOVEM.L (A7)+,ALLREGS
        RTE

rtc_int
        MOVE.L  #Hardware_RTC_Read,D0   * read the RTC to clear interrupt
        MOVE.B  #RTC_STATUS_C,D1* status register C
        TRAP    #15                     * do it
        BTST    #5,D0                   * is it alarm interrupt?
        BNE     rtc_int_alarm
        BTST    #4,D0                   * is it update interrupt?
        BNE     rtc_int_softcopy
        MOVEM.L (A7)+,ALLREGS
        RTE

rtc_int_softcopy                        * Copy the RTC values into RAM
        MOVE.L  #Hardware_RTC_Read,D0   * read the RTC seconds
        MOVE.B  #RTC_SECS,D1            * second register
        TRAP    #15                     * do it
        MOVE.B  D0,RTC_SC_SECS          * store in memory
        MOVE.L  #Hardware_RTC_Read,D0   * do the same for all the other
        MOVE.B  #RTC_SECS_ALRM,D1       * timing registers
        TRAP    #15
        MOVE.B  D0,RTC_SC_SECS_ALRM
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_MINS,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_MINS
```

```
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_MINS_ALRM,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_MINS_ALRM
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_HRS,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_HRS
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_HRS_ALRM,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_HRS_ALRM
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_DAY,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_DAY
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_DATE,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_DATE
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_MONTH,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_MONTH
        MOVE.L  #Hardware_RTC_Read,D0
        MOVE.B  #RTC_YEAR,D1
        TRAP    #15
        MOVE.B  D0,RTC_SC_YEAR
        MOVEM.L (A7)+,ALLREGS
        RTE

rtc_int_alarm
        MOVE.W  #Hardware_RTC_Read,D0   * read status register
        MOVEQ   #RTC_STATUS_B,D1
        TRAP    #15

        BCLR    #5,D0                   * clear alarm bit - disable

        MOVE.L  D0,D2                   * alarm interrupts
        MOVE.W  #Hardware_RTC_Write,D0  * write status register
        MOVEQ   #RTC_STATUS_B,D1
        TRAP    #15

        MOVE.W  #Hardware_RTC_Read,D0   * read device number
        MOVEQ   #RTC_DEVICE_ALRM,D1
        TRAP    #15

        MOVE.L  D0,D7
        MOVE.W  #Hardware_RTC_Read,D0   * read setting
        MOVEQ   #RTC_DATA_VALUE,D1
        TRAP    #15
                                        * now D7 = device to write
                                        * and D0 = value to write to it
        MOVE.L  D0,D1                   * put values in correct registers
        MOVE.L  D7,D0

        BSR     send_PIC_command        * send to PIC

        MOVEM.L (A7)+,ALLREGS
        RTE
```

```
via_int
        MOVEM.L ALLREGS,-(A7)           * Preserve all registers on stack
        MOVE.B  VIA_IFR,D0              * Read VIA interrupt flags
        BTST    #4,D0                   * Has CB1 interrupt occurred?
        BNE     via_cb1_int             * Handle it
        BTST    #2,D0                   * Has CA1 interrupt occurred?
        BNE     via_ca1_int             * Handle it
        MOVEM.L (A7)+,ALLREGS           * If neither, return
        RTE

via_ca1_int
        MOVE.B  VIA_DRA,D0              * Read register and clear interrupt
        MOVEM.L (A7)+,ALLREGS           * then return
        RTE

via_cb1_int
        MOVE.B  VIA_DRB,D0              * Read register and clear interrupt
                                        * from 6522
        MOVE.B  VIA_PCR,D0              * Read current contents of PCR
        MOVE.B  D0,D1                   * Preserve it
        BSET    #3,D0
        BSET    #2,D0                   * Take CA2 low
        BCLR    #1,D0
        MOVE.B  D0,VIA_PCR              * do it - clears PIC interrupt

        MOVEQ   #$7F,D7
via_cb1_int_delay
        DBF     D7,via_cb1_int_delay    * short delay loop

        MOVE.B  D1,VIA_PCR              * reset to original status
        MOVEM.L (A7)+,ALLREGS           * then return
        RTE

memdump
        MOVEM.L ALLREGS,-(A7)           * Preserve registers on stack
        LEA     $0,A0                   * address to start at
memdump_loop1
        MOVE.B  (A0)+,D0                * get byte from memory
        MOVE.B  D0,ACIA_TR              * transmit byte
memdump_get_tdre1
        MOVE.B  ACIA_SR,D1              * get ACIA status
        BTST    #1,D1                   * is Tx data reg empty
        BEQ     memdump_get_tdre1       * if not wait until is
        CMPA.L  #$4000,A0               * is end of ROM
        BNE     memdump_loop1           * if not loop round
        LEA     $40000,A0               * address to start at
memdump_loop2
        MOVE.B  (A0)+,D0                * get byte from memory
        MOVE.B  D0,ACIA_TR              * transmit byte
memdump_get_tdre2
        MOVE.B  ACIA_SR,D1              * get ACIA status
        BTST    #1,D1                   * is Tx data reg empty
        BEQ     memdump_get_tdre2       * if not wait until is
        CMPA.L  #$44000,A0              * is end of ROM
        BNE     memdump_loop2           * if not loop round
        MOVEM.L (A7)+,ALLREGS           * Restore stacked registers
        RTE                             * and return
```

Exception handling code

```
        ORG     $0
        DC.L    $44000                  * full descending supervisor stack
                                        * pointer
        DC.L    start                   * code address to call

        DC.L    $108                    * exception vector table
        DC.L    $10C                    * - all point to routine to set
        DC.L    $110                    * D0 to vector number then flash
        DC.L    $114                    * it on the LEDs
        DC.L    $118
        DC.L    $11C
        DC.L    $120
        DC.L    $124
        DC.L    $128
        DC.L    $12C
        DC.L    $130
        DC.L    $134
        DC.L    $138
        DC.L    $13C
        DC.L    $140
        DC.L    $144
        DC.L    $148
        DC.L    $14C
        DC.L    $150
        DC.L    $154
        DC.L    $158
        DC.L    $15C
        DC.L    $160
        DC.L    $164
        DC.L    $168
        DC.L    $16C
        DC.L    $170
        DC.L    $174
        DC.L    $178
        DC.L    $17C
trap0vector
        DC.L    trap0                  * TRAP #0 handler - OS routines
        DC.L    trap                   * Simple trap handler - just return
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
        DC.L    trap
trap15vector
        DC.L    trap15                 * TRAP #15 handler - hardware accesses

        ORG     $108                   * code to set D0 to exception
        MOVEQ   #$8,D0                 * vector number as pointed to above
```

```
BRA.S   exception
MOVEQ   #$C,D0
BRA.S   exception
MOVEQ   #$10,D0
BRA.S   exception
MOVEQ   #$14,D0
BRA.S   exception
MOVEQ   #$18,D0
BRA.S   exception
MOVEQ   #$1C,D0
BRA.S   exception
MOVEQ   #$20,D0
BRA.S   exception
MOVEQ   #$24,D0
BRA.S   exception
MOVEQ   #$28,D0
BRA.S   exception
MOVEQ   #$2C,D0
BRA.S   exception
MOVEQ   #$30,D0
BRA.S   exception
MOVEQ   #$34,D0
BRA.S   exception
MOVEQ   #$38,D0
BRA.S   exception
MOVEQ   #$3C,D0
BRA.S   exception
MOVEQ   #$40,D0
BRA.S   exception
MOVEQ   #$44,D0
BRA.S   exception
MOVEQ   #$48,D0
BRA.S   exception
MOVEQ   #$4C,D0
BRA.S   exception
MOVEQ   #$50,D0
BRA.S   exception
MOVEQ   #$54,D0
BRA.S   exception
MOVEQ   #$58,D0
BRA.S   exception
MOVEQ   #$5C,D0
BRA.S   exception
MOVEQ   #$60,D0
BRA.S   exception
MOVEQ   #$64,D0
BRA.S   exception
MOVEQ   #$68,D0
BRA.S   exception
MOVEQ   #$6C,D0
BRA.S   exception
MOVEQ   #$70,D0
BRA.S   exception
MOVEQ   #$74,D0
BRA.S   exception
MOVEQ   #$78,D0
BRA.S   exception
MOVEQ   #$7C,D0
NOP                             * Should be at $180 by now
```

```
        ORG     $180
exception
        MOVE    #$2600,SR               * Only allow NMI
        MOVE.B  D0,D4                   * preserve exception no.
        MOVE.B  D0,DOUT_H               * write exception code on high LEDs
        MOVE.B  #-1,D0                  * initial value to store on low LEDs
excep_loop
        MOVE.B  D0,DOUT_L               * store on low LEDs
        NOT     D0                      * invert D0
        MOVE.W  #$FFFF,D1               * loop variable
                                        * byte
excep_time_loop
        DBF     D1,excep_time_loop      * decrement and loop (waste time)

        BRA.S   excep_loop

excep_serial_send
        SUB     #1,D2                   * decrement D2
        CMP     #14,D2                  * is it first byte
        BEQ     excep_serial_send_excep * if so send exception number
        MOVE.B  (A7)+,D3                * get next byte of stack frame
        MOVE.B  D3,ACIA_TR              * transmit it
        BRA     excep_time_loop
excep_serial_send_excep
        MOVE.B  D4,ACIA_TR              * transmit exception number
        BRA     excep_time_loop

        ORG     trap_code

trap                                    * Temporary code to handle TRAPs
        RTE                             * Simply return

trap0                                   * Basic operating system support
                                        * D0 contains the number of
                                        * the routine to call
        CMP.B   #OS_Reset,D0            * OS_Reset
        BEQ     OS_Reset_Code           *
        CMP.B   #OS_Stop,D0             * OS_Stop
        BEQ     OS_Stop_Code            *
        RTE                             * Unknown TRAP #0

trap15                                  * Handle hardware accesses
                                        * All should be passed through here,
                                        * so that the same code will work
                                        * on different systems
                                        * D0 is the number of the routine
                                        * to be called - bits 8-15 signify
                                        * number of device being accessed.
        CMP.W   #Hardware_RTC_Read,D0
        BEQ     Hardware_RTC_Read_Code  * access real time clock
        CMP.W   #Hardware_RTC_Write,D0
        BEQ     Hardware_RTC_Write_Code * access real time clock
        RTE                             * Unknown TRAP #0

OS_Reset_Code                           * Go into SVC mode and do a reset
        RESET
        RTE                             * For completeness - will never get
                                        * here

OS_Stop_Code                            * Go into SVC mode and halt
```

```
        STOP #$0                * STOP in user mode - allow all IRQs
        RTE                             * For completeness - will never get
                                        * here

Hardware_RTC_Read_Code                  * on entry D1 = register number
                                        *               to read
                                        * on exit  D0 = register value

        MOVE.B  #$31,PIA_CRA            * select PIA DDRA
        MOVE.B  #$FF,PIA_DDRA           * make Port A outputs
        MOVE.B  #$04,PIA_CRB            * select PIA Periph Reg B
        MOVE.B  #$35,PIA_CRA            * select PIA Periph Reg A (keep CA1
                                        * interrupts enabled)
        MOVE.B  D1,PIA_PRA              * output address on Port A
        MOVE.B  #$F3,PIA_PRB            * other bits high, enable chip,
                                        * assert AS, read mode
        MOVE.B  #$F1,PIA_PRB            * negate AS - must be 135ns between
                                        * instructions for this to work
                                        * without an extra delay
        MOVE.B  #$31,PIA_CRA            * select DDRA
        MOVE.B  #$00,PIA_DDRA           * make port A inputs
        MOVE.B  #$35,PIA_CRA            * select Periph Reg A
        MOVE.B  #$F5,PIA_DDRB           * other bits high, enable chip,
                                        * read mode, assert DS
        MOVE.B  PIA_PRA,D0              * read data from Port A into D0
        MOVE.B  #$F9,PIA_PRB            * other bits high, disable RTC,read
        RTE                             * return from exception

Hardware_RTC_Write_Code                 * on entry D1 = register number
                                        *               to write
                                        *          D2 = value to write
        MOVE.B  #$31,PIA_CRA            * select PIA DDRA
        MOVE.B  #$FF,PIA_DDRA           * make Port A outputs
        MOVE.B  #$04,PIA_CRB            * select PIA Periph Reg B
        MOVE.B  #$35,PIA_CRA            * select PIA Periph Reg A (keep CA1
                                        * interrupts enabled)
        MOVE.B  D1,PIA_PRA              * output address on Port A
        MOVE.B  #$F2,PIA_PRB            * other bits high, enable chip,
                                        * assert AS, write mode
        MOVE.B  #$F0,PIA_PRB            * negate AS - must be 135ns between
                                        * instructions for this to work
                                        * without an extra delay
        MOVE.B  #$31,PIA_CRA            * select DDRA
        MOVE.B  #$FF,PIA_DDRA           * make port A outputs
        MOVE.B  #$35,PIA_CRA            * select Periph Reg A
        MOVE.B  D2,PIA_PRA              * write data from D2 to Port A
        MOVE.B  #$F4,PIA_PRB            * other bits high, enable chip,
                                        * write mode, assert DS
        MOVE.B  #$F9,PIA_PRB            * other bits high, disable RTC, read
        RTE                             * return from exception
```

Definitions file

```
* Definitions for 68k based home automation system

* Register sets

ALLREGS REG     D0-D7/A0-A6             * Register set for all registers
DATAREGS        REG     D0-D7           * Data register set
ADRREGS         REG     A0-A6           * GP Address registers
ALLADRREGS      REG     A0-A7           * All Address registers

* Hardware addresses

IO_BASE         EQU     $80000          * Allow shifting of IO space

ACIA            EQU     IO_BASE+$000
ACIA_SR         EQU     IO_BASE+$000    * Status register  - read only
ACIA_CR         EQU     IO_BASE+$000    * Control register - write only
ACIA_RR         EQU     IO_BASE+$002    * Receive register - read only
ACIA_TR         EQU     IO_BASE+$002    * Transmit register- write only
SER_BPS         EQU     IO_BASE+$004    * Serial BPS and control lines

PIA             EQU     IO_BASE+$200
PIA_PRA         EQU     IO_BASE+$200    * Peripheral register A (CRA bit 2 set)
PIA_DDRA        EQU     IO_BASE+$200    * Data direction register A (CRA bit 2 clear)
PIA_CRA         EQU     IO_BASE+$202    * Control register A
PIA_PRB         EQU     IO_BASE+$204    * Peripheral register B (CRB bit 2 set)
PIA_DDRB        EQU     IO_BASE+$204    * Data direction register B (CRB bit 2 clear)
PIA_CRB         EQU     IO_BASE+$206    * Control register B

VIA             EQU     IO_BASE+$300
VIA_DRB         EQU     IO_BASE+$300    * Data register B
VIA_DRA         EQU     IO_BASE+$302    * Data register A
VIA_DDRB        EQU     IO_BASE+$304    * Data direction register B
VIA_DDRA        EQU     IO_BASE+$306    * Data direction register A
VIA_T1CL        EQU     IO_BASE+$308    * T1 Low counter
VIA_T1CH        EQU     IO_BASE+$30A    * T1 High counter
VIA_T1LL        EQU     IO_BASE+$30C    * T1 Low latches
VIA_T1LH        EQU     IO_BASE+$30E    * T1 High latches
VIA_T2CL        EQU     IO_BASE+$310    * T2 Low counter
VIA_T2CH        EQU     IO_BASE+$312    * T2 High counter
VIA_SR          EQU     IO_BASE+$314    * Shift Register
VIA_ACR         EQU     IO_BASE+$316    * Auxiliary Control Register
VIA_PCR         EQU     IO_BASE+$318    * Peripheral Control Register
VIA_IFR         EQU     IO_BASE+$31A    * Interrupt Flag Register
VIA_IER         EQU     IO_BASE+$31C    * Interrupt Enable Register
VIA_NHDRA       EQU     IO_BASE+$31E    * No Handshake DRA

DIN             EQU     IO_BASE+$600    * Digital In
DOUT            EQU     IO_BASE+$600    * Digital Out
DOUT_H          EQU     IO_BASE+$600    * Digital Out - high byte
DOUT_L          EQU     IO_BASE+$601    * Digital Out - low byte

RTC_SECS        EQU     $00             * Real Time Clock internal addresses
RTC_SECS_ALRM   EQU     $01
RTC_MINS        EQU     $02
RTC_MINS_ALRM   EQU     $03
RTC_HRS         EQU     $04
RTC_HRS_ALRM    EQU     $05
```

```
RTC_DAY          EQU     $06
RTC_DATE         EQU     $07
RTC_MONTH        EQU     $08
RTC_YEAR         EQU     $09
RTC_STATUS_A     EQU     $0A
RTC_STATUS_B     EQU     $0B
RTC_STATUS_C     EQU     $0C
RTC_STATUS_D     EQU     $0D
RTC_DATE_ALRM    EQU     $0E
RTC_MONTH_ALRM   EQU     $0F
RTC_YEAR_ALRM    EQU     $10

RTC_DEVICE_ALRM EQU      $0E
RTC_DATA_VALUE   EQU     $0F

* Software interrupt numbers

OS_Reset         EQU     $0                      * SWI number
OS_Stop          EQU     $1

Serial_GetByte   EQU     $0
Serial_PutByte   EQU     $1
Hardware_RTC_Write       EQU     $900
Hardware_RTC_Read        EQU     $901

* RAM allocations

RAM_BASE         EQU     $40000          * Allow shifting of RAM space

SERIAL_BUFLEN    EQU     $FF             * Size of receive and transmit buffers
SERIAL_STATUS    EQU     RAM_BASE+$000   * Store serial status on IRQ
SERIAL_RX_INPTR EQU      RAM_BASE+$002   * Pointer to store incoming data at
SERIAL_RX_OUTPTR         EQU     RAM_BASE+$004   * Pointer to read from
SERIAL_TX_INPTR EQU      RAM_BASE+$006   * Pointer to write to
SERIAL_TX_OUTPTREQU      RAM_BASE+$008   * Pointer to outputting data
SERIAL_RX_OVERFLOW       EQU     RAM_BASE+$00A   * Flag to show if overflows
SERIAL_TX_OVERFLOW       EQU     RAM_BASE+$00C   * Flag to show if overflows
SERIAL_CONFIG    EQU     RAM_BASE+$00E
SERIAL_WORKSPACE         EQU     RAM_BASE+$980   * Workspace for string processing
SERIAL_INBUF     EQU     RAM_BASE+$1000  * Serial input buffer
SERIAL_OUTBUF    EQU     SERIAL_INBUF+SERIAL_BUFLEN      * Serial output buffer
SERIAL_INEND     EQU     SERIAL_OUTBUF-2
SERIAL_OUTEND    EQU     SERIAL_INBUF+SERIAL_BUFLEN - 2

RTC_SC_SECS      EQU     RAM_BASE+$010   * Soft copy of Real Time Clock
RTC_SC_SECS_ALRM         EQU     RAM_BASE+$011
RTC_SC_MINS      EQU     RAM_BASE+$012
RTC_SC_MINS_ALRM         EQU     RAM_BASE+$013
RTC_SC_HRS               EQU     RAM_BASE+$014
RTC_SC_HRS_ALRM EQU      RAM_BASE+$015
RTC_SC_DAY               EQU     RAM_BASE+$016
RTC_SC_DATE      EQU     RAM_BASE+$017
RTC_SC_MONTH     EQU     RAM_BASE+$018
RTC_SC_YEAR      EQU     RAM_BASE+$019

TF_BLK           EQU     RAM_BASE+$1200  * Block to hold timeframes
TF_BLK_END       EQU     RAM_BASE+$2000  * allow plenty of space
TF_START         EQU     RAM_BASE+$020   * Pointer to first
TF_END           EQU     RAM_BASE+$022   * Pointer to last
```

```
TF_TIME         EQU     $0              * Offset into timeframe
TF_DEVICE       EQU     $4
TF_ACTION       EQU     $6
TF_LEN          EQU     $A              * Length of timeframe block

* Miscellaneous definitions

LF              EQU     10
FF              EQU     12
CR              EQU     13
```

Bibliography and References

Motorola, “Motorola Semiconductor Master Selection Guide”, Tenth Revision, Motorola Inc., Phoenix, Arizona, 1996.

Motorola Semiconductor Products Sector,“M68000 User’s Manual”, Ninth Edition, Motorola Inc., Tempe, Arizona, 1994.

Motorola, “M68000 Family Programmer’s Reference Manual”, First Revision, Motorola Inc., Phoenix, Arizona, 1992.

Mike Tooley, “Newnes 68000 Family Pocket Book”, Newnes, Oxford, 1992.

Mike Tooley, “Newnes Computer Engineer’s Pocket Book”, Fourth Edition, Newnes, Oxford, 1996.

Motorola Logic Integrated Circuits Division, “FAST and LS TTL Data”, Fifth Edition, Motorola Inc., Phoenix, Arizona, 1992.

National Semiconductor, “National Application Specific Analog Products Databook”, 1995 Edition, National Semiconductor Corporation, Santa Clara, California, 1995.

Paul Horowitz, Winfield Hill, “The Art of Electronics”, Second Edition, Cambridge University Press, Cambridge, UK, 1989.

Acorn Computers Ltd, “Acorn Risc PC Welcome Guide”, Issue 1, Acorn Computers Ltd, Cambridge, UK, 1994.

M. W. Brimicombe, “Electronic Systems”, Thomas Nelson and Sons, Walton-on-Thames, UK, 1985.

Mark Ward, “Digital data comes down the mains”, Technology, *New Scientist*, 18 January 1997.

“Remote control of your home energy”, *The Messenger*, 12 February 1997.

Internet comp.home.automation newsgroup frequently asked questions list (X10 section).

Data sheets on MC68000, MC146818A, MC6821, MC6840, MC6845, MC6850, MC6854, W65C22S, MAX1691, MAX665, MAX1232, MAX238CNG, R6502, PIC16C84, OP282.

Amar Palacherla, “Application Note AN555: Software Implementation of Asynchronous Serial I/O for PIC16CXX”, Arizona Microchip, Chandler, Arizona, 1995.

Acknowledgements

Semiconductor data generously supplied free of charge by Motorola Inc., Maxim Integrated Products (UK) Ltd, The Western Design Center, Inc. (Mesa, Arizona), Analog Devices Inc., Arizona Microchip Corp., United Microsystems Corp., Rockwell Inc., and National Semiconductor Corp.

Free IC samples supplied by Maxim Integrated Products (UK) Ltd.

68000 assembler and simulator software written and made freely available by Paul McKee and Marwan Shaban, North Carolina State University.

PIC programmer hardware design and programming software by David Tait, Manchester University.

RS Components
Rapid Electronics
Farnell Components
HB Electronics

Mr LM Holland
Mr ATJ Evans