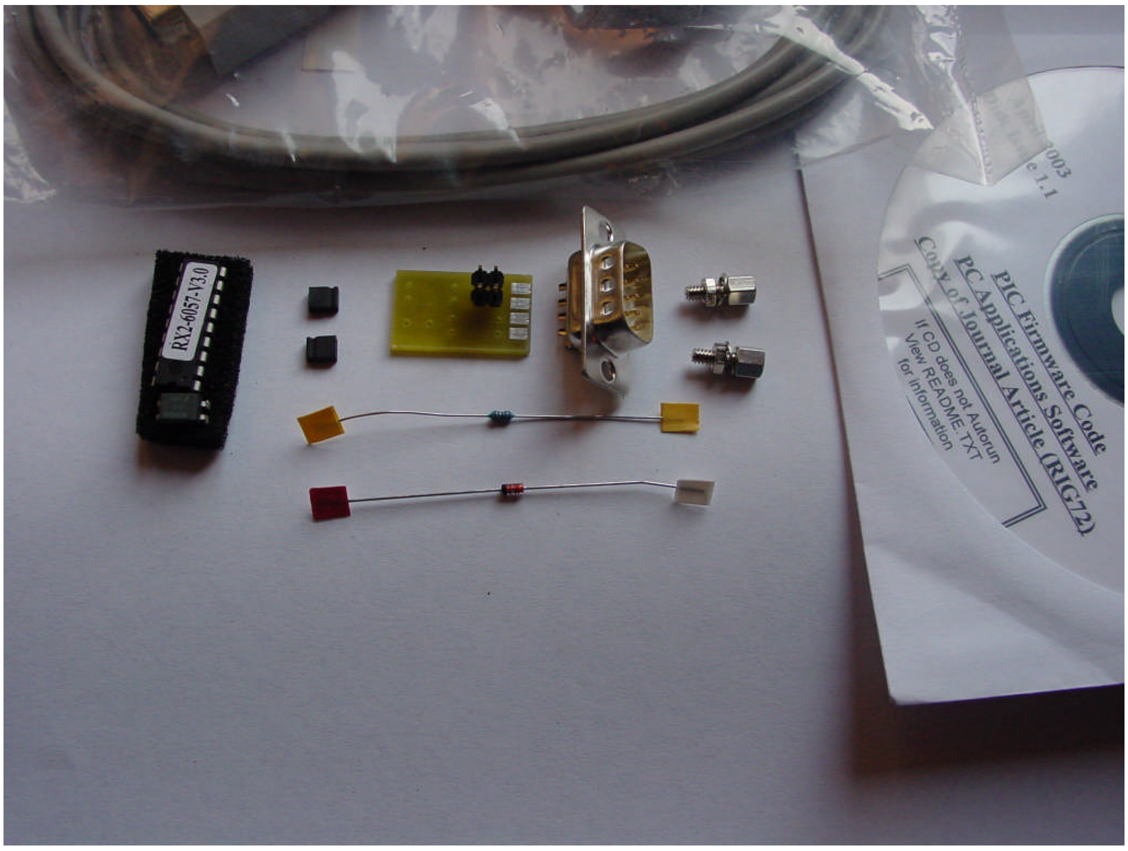
SERIAL REMOTE CONTROL

FOR THE RX2



Updated May 2005 to support NOAA-18

Introduction

Some years ago, Max Hadley published an article detailing how enthusiasts could construct a simple serial control unit that enabled the RX2 receiver to be controlled directly from a personal computer.

In the pages that follow, we describe the kit and we reprint Max Hadley's original article for those who want to 'go it alone'.

The Kit – The Components

- a D-type male 9-pin connector
- the requisite jack posts for this connector
- a small printed circuit board for the interface components
- components for this PCB namely, a 2k7, 0.4W resistor, a 1N4148 diode, an SFH6l8-4 opto isolator, and two option setting links (see later)
- a new PlC microprocessor to replace the one already fitted in your RX2
- a cable to connect from the COM port of your PC to this interface
- a CD-ROM with David Taylor's PassControl and WXtrack (unregistered version), Max's source code and compiled HEX file for the PlC, plus
- re-prints of the relevant RIG Journal articles, and build instructions for the interface

Options

There are two types of 9-pin PC serial cables. The one supplied with this kit is wired pin-2 to pin-3, pin-3 to pin-2 and pin-5 to pin-5. This is known as a null modem cable. The other common type is an extension cable that is wired pin-2 to pin-2, pin-3 to pin-3 and pin-5 to pin-5.

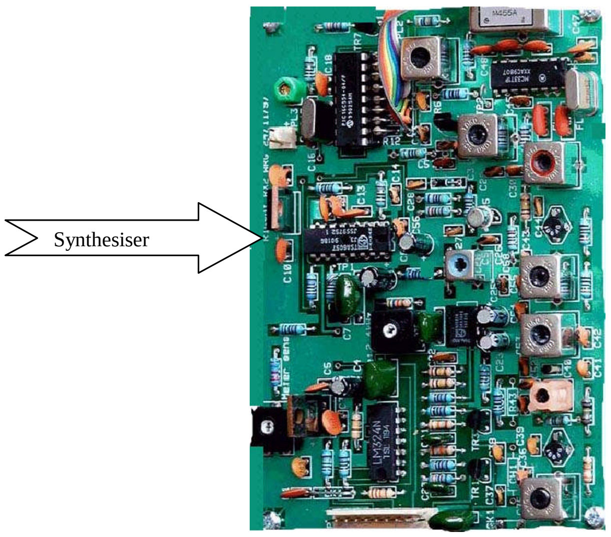
If the cable supplied is, for example, not long enough, and you are unable to source a null modem cable of the required length then the option has been fitted into the PCB to allow the use of the straight-through type.

The PCB has two 2-pin header' connectors and one moveable shorting jumper. For null modem cables' (as supplied) the jumper should be fitted to the header plug marked '2' in the copper of the PCB (2 being the input pin number on the D-type connector), whereas for the straight through cable type it should be fitted to the header plug marked 3 (you guessed it – input Pin-3 on the D-type).

You cannot damage the interface if the jumper is in the wrong position - it just will not work - so trial and error is fine if you have a cable and you are unsure which type it is.

The Synthesiser

Early RX2's were fitted with the TSA6060 synthesiser IC. The vast majority are fitted with the TSA6057. The new PIC processor included in the kit is set for the TSA6057. If you have an early RX2, or if you are in doubt at all, please check 1C4 in your RX2.

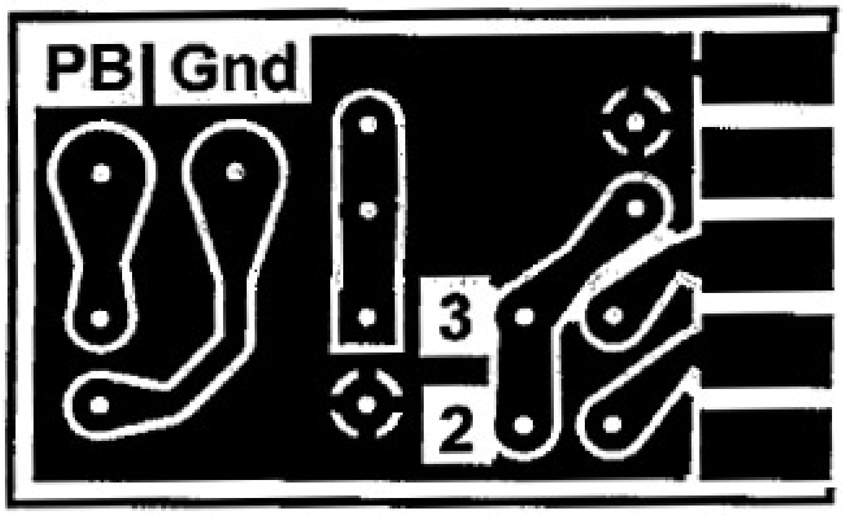


If it is not a TSA6057 then please contact the RIGShop for an exchange of PIC.

The PICs supplied, are 'flash' parts - i.e. they are re-programmable at a later date if new frequencies or features become available. If this is required in the future there would be a nominal charge from the RIGShop for this service.

Assembly and Connection

The printed circuit board is very simple to make. No components on the interface PCB are static sensitive - but please see later when working on the RX2.



Top view of the Remote Control PCB

Push the D-type connector on to PCB so that the row of five contacts on the connector line up with the five pads on the PCB (and on the underside the row of four pins line up" with the corresponding four pads on the PCB). The PCB should be pushed in until it is in contact with the main body of the connector (see the photograph) and soldered in place.

You now have a slightly larger assembly to handle - the PCB on its own is quite small and fiddly to work with.

Fit the opto-isolator, ensuring that it is the correct way round.

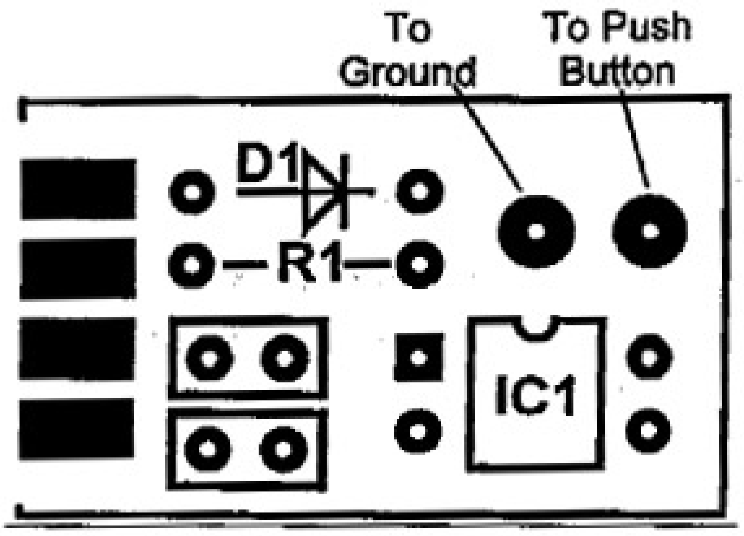
Pin-1 on the PCB is marked by having a square pad, and is closest to the two option headers. Pin-1 on the actual component is marked by the side with the groove (as opposed to the more usual methods of having a dimple adjacent to pin-1). Double-check before soldering.

Fit the resistor and diode. **Check diode polarity before soldering** (black bar away from the connector). Finally fit the two 2-pin header plugs, long pins upwards. A piece of masking tape is useful for holding them in place whilst soldering, otherwise they tend to fall out. Now you have completed the PCB assembly.

Wiring the PCB to the RX2 Board

> ***From now on be certain to observe, anti-static precautions - ensure that both you and your tools are at earth potential and that the RX2 is powered off.***

The black wire goes from the ‘Gnd’ pad on the interface to any convenient ground point on the RX2 board (scrape the green resist off the ground plane adjacent to R4 and solder it down there). The white wire goes from the push-button (PB) pad on the interface to the PB connection on the RX2 (1C6 pin-1 R4 and PL1 pin-9. I soldered to the end of R4 closest to the edge of the board. Beware if you solder to the front-panel push button itself - it melts very easily!



Bottom view of the Remote Control PCB

Now carefully remove the original PIC from your RX2 (1C6). If you do not have a proper IC 'puller' to gently lever each end in turn with a small screwdriver until it pops out of its socket.

Carefully align the new PIC, ensuring that it's the correct way round before gently pushing it into the socket. Check no pins have been forced out or under the socket. That's it!

Less than half an hour's work.

Troubleshooting

There really is not much to go wrong, but if all else fails try the following –

- Are you using the correct COM port on your PC?
- Are the opto-isolator and diode correctly orientated on the interface PCB?
- Have you set the option jumper correctly for the cable you are using?
- Have you connected the black and white cables correctly at both the interface and RX2 ends?
- Do you have the correct PIC for your synthesiser?
- Is the PIC inserted the socket correctly, with no bent pins?

If you find that noise from the PC is affecting your images (which is unlikely) it is worth using a longer PC to interface cable that allows the RX2 to be moved farther away from the computer. Another helpful trick is to attach a clip-on ferrite to the cable (try different locations along the cable).

Credits

One last word, don't forget this would not be available without the hard work David Taylor has put in to his software - so if you enjoy and use Wxtrack please say thank you by registering it with him.

- Max Hadley for permission to use his original design;
- Sam Elsdon for the port of the code to the PIC16F84A;
- Clive Finnis for the original work on production of this kit; and
- the original RX2 development team.

Serial Remote Control for the RIG RX-2 Receiver (as printed in Journal 60)

Max Hadley

This article describes the modifications to the RIG RX-2 receiver that enable it to be controlled remotely from a computer, using an RS-232 serial interface. The modifications are in two parts: some simple additional hardware, and a new firmware program for the PIC16C84 micro-controller.

Operation

In manual control mode, the modified receiver operates very much as the original. When the RX-2 is switched on, it will emit a 1 second beep, during which all the display segments are turned ('lamp test'). This allows time for the signal detector circuit to settle. After the settling period, it will start scanning from channel 1, stopping only when it finds a signal. When a signal is found, the RX-2 emits a longer beep and stops scanning, until 15 seconds after the signal is lost. It then emits a short beep and re-starts scanning. A short button press at any time will jump to the next channel and stop scanning, accompanied by a very short beep. A longer press will re-start scanning, with a longer beep.

At the same time, any characters that arrive at the serial port are checked for valid commands. There are two valid commands: 's' (or 'S' - case is not important) and 'F'. The 'F' command, followed immediately by a channel number 0 to 9, will change to that channel, and stop scanning, if a scan was in progress. 'S' will re-start scanning. No carriage return or line feed is required after the command. Any incomprehensible or extra characters are quietly ignored.

The remote control facility increases the RX2 channel capacity to 10 – allocated as follows:

Channel Number	**Frequency (MHz)**
0	137.200
1	137.100
2	137.400
3	137.500
4	137.620
5	137.910
6	137.300
7	137.700
8	137.800
9	137.850

The channel allocation reflects the order used by David Taylor in his PassControl software.

A button push, or an 'S' command, when the receiver is tuned to one of the 'extra' channels, will re-start scanning from channel 1.

The serial interface is designed to make the minimum demands on the computer driving it. It does not use any flow control: it operates at a fixed 1200 baud, and can handle continuous characters at this speed. There are no 'handshaking' signals. It cannot even send any data to the computer!

PIC Software

The source code for the modified PIC program is available at:

http://www.susato.demon.co.uk/RX2_C84_src.zip

This requires Microchip MPLAB/MPASM to assemble, and a suitable device programmer for the PIC16C84. For convenience, the Intel HEX format ready for the programmer is available at:

http://www.susato.demon.co.uk/RX2_C84.HEX

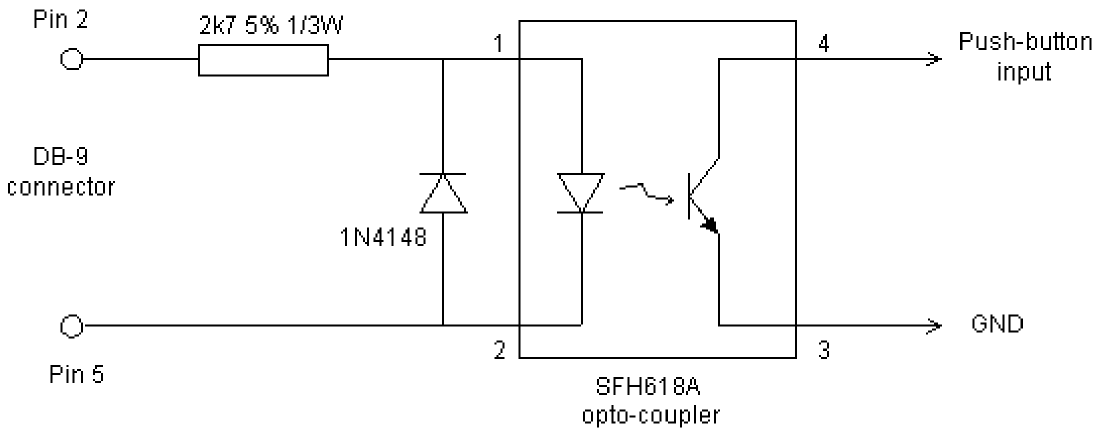
The software operation is based on the idea that when a person presses the button, it stays 'down' for a long time - at least a long time compared to the duration of a serial character. The serial input and push button use the same pin. The program waits for a 'low' logic level on this pin. This could either be the start of a serial character or the start of a button push. The software first attempts to decode the signal as a character. If the input is still low at the end of the character period, when it ought to be high, the software decides that it has a button push. Two separate tasks handle the button pushes and decode the received characters.

A more detailed explanation of the software is included latter in this article.

Hardware Modifications

The hardware modifications have been kept as simple as possible. The serial interface uses the same PIC input pin currently used by the push-button, and can be operated in conjunction with the button. It requires no extra power supply, drawing what power it needs from the PC and through the pull-up resistor on the push-button input.

The complete circuit diagram is at **Figure 1.**



In the RS-232 'idle' state, DB-9 pin 2 will sit at a voltage of -5V to -12V. The opto-coupler LED will be off, and so will be the transistor. Pin 4 will be pulled up to +5V by the push-button pull-up resistor on the RX-2 circuit board. A button push will take this point low to GND, unaffected by the presence of the serial interface.

When an RS-232 character comes along, the start bit will take DB-9 pin 2 to between +5V and +12V, turning on the opto-coupler LED and transistor. The push-button input will go low, to be detected by the software.

The push button input goes to pin 9 of SK1 on the RX-2: GND is found on pin 8 of the same connector. DB-9 connector pin 5 is ground, pin 2 is Rx-data. You will need to use a 'null modem' cable with these connections. Alternatively, wire the resistor to pin 3 and use a 'straight through' cable.

You can also use a 4N25 or similar opto-isolator, the exact device is not critical. For the 4N25, GND goes to pin 4 and the push-button input to pin 5. Pins 3 and 6 are no-connect. Using an opto-isolator helps to keep electrical noise from the computer out of the receiver.

Construction

Construction technique is not critical. I built the prototype hardware on a very small piece of Veroboard which I fixed to the back of the DB-9 socket. The output is best wired straight to the push-button, using stranded hook-up wire. Take care when desoldering the existing wiring from the push-button, as the supplied parts tend to melt rather easily. If you are at all in doubt of your skills, order a replacement push-button along with the other parts! No serious harm will be caused if you get the two wires to the button crossed, but probably neither the button nor the serial input will work until you correct the situation.

Notes

1. If you intend using this modification with a computer running Microsoft Windows NT, you should be aware that as it boots, this operating system sends a sequence of 'breaks' to each serial port it finds. These are interpreted as button pushes, causing the RX-2 to switch channels and stop scanning. A long button push - or of course an 'S' command - will restart the scan.

The Software – a more detailed explanation

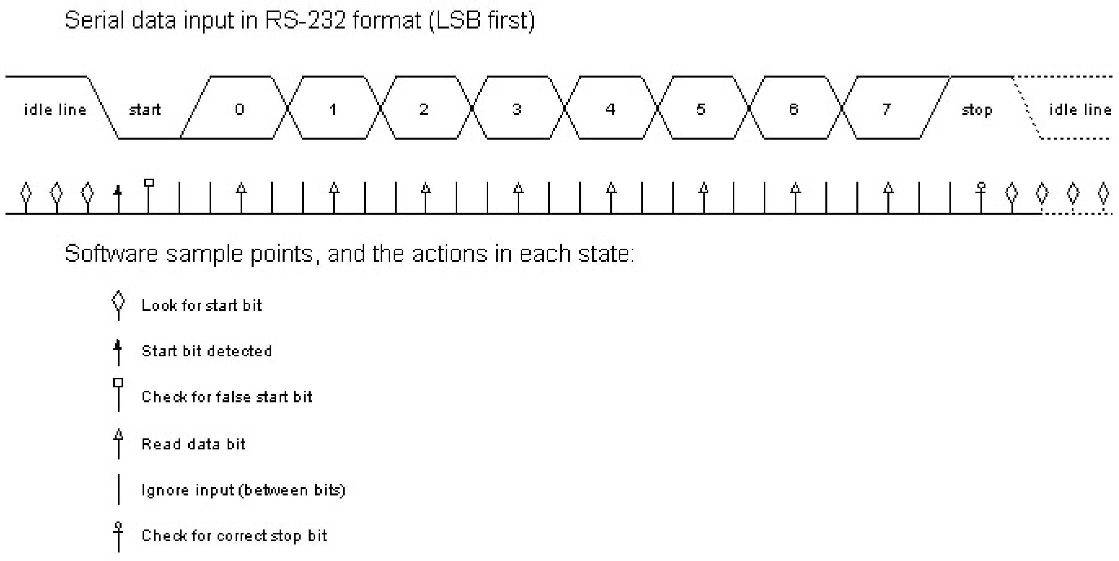
When I first studied the original source code for the RX2 by Steve Drury, I realised it was organised around a main loop. At the top of this loop, a short stretch of code waits for a fixed time period to expire, before calling a sequence of routines that do the actual work. The loop executes at a rate (set by the timer period value) of 3,500 Hz. This rate sets, among other things, the frequency of the 'beep' - 1,750Hz.

Now 3,500Hz isn't far removed from 3,600Hz, which is three times the bit rate for a 1200-baud asynchronous serial data link. If the loop were adjusted to run at this frequency, it would be possible to sample the push-button input at this rate. This would allow a 'software UART' to decode characters sent from a computer. I also realised that it would be possible to share the input pin between the push-button and the serial data. At the end of each serial character, the asynchronous protocol requires a stop bit to be sent. For one bit period the input pin must be high. This allows button pushes to be detected, as the input pin will still be low at that point.

The main_loop, starting at line 936, begins by delaying until the timer TMR0 times out, then immediately reloads it with a value to give a 3,600Hz repetition rate (278 microseconds). Each of the 7 tasks is called in turn, then a 16-bit system timer (counting time in 278 microsecond chunks) is incremented, and the loop repeats. So long as the total run-time of all 7 tasks does not exceed the total time available, each will run at a regular rate. This type of organisation is called a cyclic executive.

The first task, beep_pin_driver_task, flips the beep output pin each time round, when a beep is required, and does nothing otherwise. Thus one difference between this code and the original is that the pitch of the beep is raised to 1,800Hz. Beeping is controlled by the location beep_mask. A call to one of the 'beep' subroutines (lines 740-746) sets this mask, and loads a time-out location with a value depending on the length of the required beep. The 7th task, beep_task, sets beep_mask back to zero when the time-out expires.
The second task in the schedule is serial_task, which looks for serial characters on the input pin. This is implemented as a table-driven state machine. It samples the input pin at three times the baud rate, initially looking for a start bit (low input

level). Now when a low level is detected, all we know about the exact time at which the input line went low is that it is somewhere between the last two input sampling instants, three of which occur per bit time. If we assume that, on average, the real start of the start bit was half-way between these two samples, the middle of the start bit will be one third of a bit period after the second sample, i.e. at the next sample time. Confused? you should be. **Figure 2** may help:


Serial data input in RS-232 format (LSB first)
idle line
start
0
1
2
3
4
5
6
7
stop
idle line
Software sample points, and the actions in each state:
Look for start bit
Start bit detected
Check for false start bit
Read data bit
Ignore input (between bits)
Check for correct stop bit

In the idle line state, serial_task is sampling the input line, looking for a start bit. Eventually it detects one. At the next sampling instant, it checks that the start bit is still there, as it should be. If it has gone away, serial_task assumes that the 'start bit' was just a noise spike, and goes back to looking for a real start bit. If the start bit was valid, then every three sample periods (i.e. once per bit period) one bit of the character is read in, starting with the LSB. After 25 sample periods, all eight bits have been read in. Now comes the crunch. One bit time later, the input must be high for a valid character. If it is still low, we have a button push; if high, a character. If the character is valid, it is copied to location ser_data and the RD flag is set. If not, flag BD is set.

Tasks input_task and button_task handle serial characters and button pushes respectively. Each waits in an idle state until the appropriate flag is set by serial_task. Each implements a state machine that decodes the input (in terms of characters and button push durations) and translates it into control information.

When the receiver is scanning, scan_task is responsible for changing frequency, detecting a satellite signal, and implementing the AOS and LOS time-outs. A call to StartScanning (line 849) will wake this task from its idle state and begin the scanning operation. Scanning is stopped by forcing the state of scan_task to 0 (e.g. line 646). While scanning, this task updates the LED display and tunes the receiver autonomously.

When the receiver is not scanning, either input_task or button_task must control the display and tune the receiver directly. Controlling the display is simple enough - a call to UpdateDisplay does that - but there is a complication when tuning the receiver. The I2C bus used to program the synthesizer chip is fairly slow, and it is not possible to send all the required data to the chip in the 278 microseconds available.

To get round this, a separate task I2C_task is responsible for programming the synthesizer. This task idles until told to send new data. It then translates the current channel number into programming data, and sends the data at the rate of one byte per 278 microsecond 'timeslice'. When this task is running, it takes by far the largest fraction of the 278 microseconds available! A minor complication arises because the vital frequency data must be sent in two bytes. Between the two bytes another task could slip in and change the channel number, so the synthesizer would get the high byte corresponding to one channel and the low byte from another. To get round this I2C_task keeps a local copy of the channel number, and uses this. Before going back to the idle state, it checks to see whether the 'real' channel is still the same as its local copy, and if not, it runs itself again with the new, changed channel.

Any task can produce a beep (and several do) by calling one of the 'beep' subroutines (lines 740-746). The beep_task will take care of all the timing involved.
At power-on, after initialising the system, the software checks for a low level on the button input pin. If the pin is low, the receiver alignment 'channel 11' (137.97MHz) is set, and serial_task is placed in a special 'black hole' state from which there is no exit. Since neither the RD nor BD flag will ever be set, the only way out of this situation is a power-on reset.

Under more normal circumstances, channel 1 is set and a 1 second beep initiated. My initial implementation jumped straight to main_loop at this point. However, the tone decoder used to detect the 2.4kHz subcarrier always gives a false detection just after power on, and this caused a false AOS. To get round this, the code now jumps to power_on_loop, which runs only beep_pin_driver_task, I2C_task, and beep_task. Extra code at the start of this task monitors the status of beep_mask and jumps to main_loop when the beep finishes. The 1 second beep gives adequate time for the tone decoder to settle.

During this period all LED segments are turned on as a confidence check.