```
/*****************************************************************************

   YAPP - Yet Another PIC Program

   YAPP.c

   Copyright © Martin Newell, 2005,2006 San Jose, California
   Permission granted for personal use only
   Use for commercial purposes prohibited without permission

 **    Updated Dec 24, 2005

 * Option to support servo-style PCM on throttle to drive an electronic speed controller, ESC, instead
   of the pulse-coded brushed motor output.
   This was added to be able to drive a 3-phase brushless motor controller.
   This form of motor output is not controlled by the stand-by procesdure.  Behavior on
   loss of signal is determined by the external electronic speed control.
   See #define MOTOR_PWM

 * New output pin option, see #define RUDDER_RUDDER
   Outputs rudder on pairs of pins to give 50mA drive off the chip which can drive >=80 ohm actuator directly.
   Uses pins 2,3 connected and 6,7 connected.  Motor is output on normal pin 5.
   Intended for an add-on chip to add 4th channel rudder control.
   Can be used in combination with MOTOR_PWM (see above) to give 3 actuators and PWM motor output.

 **    Original Version June 17, 2005

 * For PIC12F629/675 and PIC12F635/683

 * With PIC12F629 (4MHz), input is read to resolution of 10 for left, right, up, down and 20 for throttle
   With PIC12F635 (8MHz), input is read to resolution of 20 for left, right, up, down and 40 for throttle but the
       throttle reading is divided by 2 since output resolution is 20 in all cases
   Output resolution of 20 for aileron and elevator is used even for 4MHz to give finer resolution in expo table.
   Input resolution for 4MHz could be doubled by using the GPIORead and OCHalfTimer variables, but currently not

 * Uses Pulse Frequency Modulation, PFM
   Output frequency for PIC12F629(4MHz) is 5.26 KHz to 26.3 Khz at mid range
   Output frequency for PIC12F635(8MHz) is 5.26 KHz to 52.6 Khz at mid range

 * Decodes 4 channels
   4th channel could be used in separate chip - see RUDDER_ELEVATOR and RUDDER_RUDDER below

 * Adapts to  positive- or negative-going input

 * Adapts to transmitter channel configuration.
   Expects aileron, (which is typically used to control the rudder), elevator and rudder to be on increasing channels.
   with throttle channel inserted anywhere. Throttle taken to be channel furthest from center on start up.

 * Supports Rudder/Aileron-Elevator and Elevon modes.
   Attach left elevon to aileron outputs with up for left,
       and right elevon to elevator outputs with up for up.
   See "Startup Configuration" below to activate Elevon mode.

```

```
54      * Uses "Startup Configuration" period while throttle stick above zero.  Motor will not start.
55
56        Start Rx with Throttle at about 3/4 max
57        Startup Configurations include:
58        1. Move Aileron/Elevator stick to top right corner for 1/2 second => sets Elevon mode
59        2. Move Aileron/Elevator stick to top left corner for 1/2 second => un-sets Elevon mode
60        3. Move Aileron/Elevator stick to bottom right corner for 1/2 second => sets negative Expo mode
61        4. Move Aileron/Elevator stick to bottom left corner for 1/2 second => un-sets negative Expo mode
62        5. Move Throttle stick to max for 1/2 second => callibrates Rx to neutral at current Aileron, Elevator
63            and Rudder stick positions
64        6. Move Throttle stick to minimum for 1/2 second => Enables motor and all configurations are locked.
65        Control surfaces will twitch when a mode is changed, for confirmation.
66        Configuration is remembered in EEPROM and is restored on next power up, except for Center Callibration.
67
68        Normal operation, after configuring:
69        Start Rx with Throttle at minimum => Saved modes are restored and throttle is enabled, or,
70        Start Rx with Throttle at maximum => Rx callibrates center to Tx sticks.  Move throttle to minimum to fly.
71
72
73      * Goes through progressive standby if signal lost, until found again
74        This requires finding inter-frame gap and some number of valid frames
75        1. After 1.5 seconds - Aileron goes 1/5 left, Elevator to neutral
76        2. After another 1.5 seconds - Throttle multiplied by 3/4
77        3. Repeat 2. until Throttle zero
78
79      * Uses polling, no interrupts
80
81      * This file reads best with 4 character tabs, fixed pitch font
82
83    ************************************************************************/
84
85    // Enable only one of the following
86    #define PIC12F629                           // 8 pin,, 4MHz, OSCCAL, EEDATA
87    //#define PIC12F635                         // 8 pin, 8MHz, OSCTUNE, EEDAT
88    //#define PIC12F675                         // 8 pin,, 4MHz, OSCCAL, EEDATA, A/D
89    //#define PIC12F683                         // 8 pin,, 8MHz, OSCTUNE, EEDAT, A/D
90
91
92    /************************* Common Declarations ***************************/
93
94    #ifdef PIC12F629
95    #define CLOCK_FREQ 4                        // MHz
96    #define TICK 38                             // One tick of OCTimer - units of uS
97    #pragma config = 0x3F84
98    #endif
99
100   #ifdef PIC12F635
101   #define CLOCK_FREQ 8                        // MHz
102   #define TICK 19                             // One tick of OCTimer - units of uS
103   #pragma config = 0x33C4
104   #pragma bit IRCF0   @ OSCCON.4                // definitions missing from PIC12F635.h
105   #pragma bit IRCF1   @ OSCCON.5
106   #pragma bit IRCF2   @ OSCCON.6
```

```
107  #endif
108
109  #ifdef PIC12F675
110  #define CLOCK_FREQ 4                        // MHz
111  #define TICK 38                             // One tick of OCTimer - units of uS
112  #define HAS_AD
113  #pragma config = 0x3F84
114  #endif
115
116  #ifdef PIC12F683
117  #define CLOCK_FREQ 8                        // MHz
118  #define TICK 19                             // One tick of OCTimer - units of uS
119  #define HAS_AD
120  #pragma config = 0x33C4
121  #pragma bit IRCF0   @ OSCCON.4                 // definitions missing from PIC12F683.h
122  #pragma bit IRCF1   @ OSCCON.5
123  #pragma bit IRCF2   @ OSCCON.6
124  #endif
125
126  //#define SIMULATOR                         // Comment out for release code
127
128  #pragma update_RP 0                         // Disable setting RP0 except where need in initialization
129
130  // Chip Configuration parameters
131  // Oscillator Internal
132  // Watchdog Off
133  // Power Up Timer On
134  // MCLR Internal
135  // Brown Out Off
136  // Code Protect Off
137  // Data EE Read Protect Off
138
139  //#pragma optimize 1
140
141  char pulse;                                 // Sense of timing pulses.  Set to INPUT_PIN or 0 for +ve or -ve going pulses
142  #define RESOLUTION      20                  // Defines number of steps from 0 to max for each channel
143  #define INPUT_PIN       8                   // Used instead of bit addressing for better code
144  #define INPUT_ON        ((GPIO & INPUT_PIN) == pulse)   // value of 'pulse' defines whether On = high or low
145  #define INPUT_OFF       ((GPIO & INPUT_PIN) != pulse)   // value of 'pulse' defines whether Off = low or high
146  char MotorChannel;                          // The motor channel [0..3]
147  //#define MOTOR_PWM                         // Generate servo-style PWM on motor pin instead of PFM for direct motor drive
148
149  // Define bits in GPIO
150  // Exactly one of the following 4 blocks must be enabled by un-commenting the initial #define
151
152  #define AILERON_ELEVATOR                    // Conventional Aileron-Elevator (e.g. Leichty)
153  //
154  //          Vcc -|1     8|- Ground
155  //     Elevator -|2     7|- Aileron
156  //     Elevator -|3     6|- Aileron
157  //        Input -|4_____5|- Motor
158  //
159  #ifdef AILERON_ELEVATOR
```

```
160   #define AILERON_LEFT            1           // GP0 Aileron Left
161   #define AILERON_RIGHT           2           // GP1 Aileron Right
162   #define MOTOR                   4           // GP2 Motor
163   #define ELEVATOR_DOWN           0x10        // GP4 Elevator Down
164   #define ELEVATOR_UP             0x20        // GP5 Elevator Up
165   #define RUDDER_LEFT             0           // Not output Rudder Left
166   #define RUDDER_RIGHT            0           // Not output Rudder Right
167   #define INVERT_OUTPUT_MASK      0           // No inversions
168   #endif
169
170   //#define RUDDER_ELEVATOR                   // For auxiliary 4th channel chip
171   //                  _______
172   //          Vcc -|1     8|- Ground
173   //     Elevator -|2     7|- Rudder
174   //     Elevator -|3     6|- Rudder
175   //        Input -|4_____5|- Motor
176   //
177   #ifdef RUDDER_ELEVATOR
178   #define RUDDER_LEFT             1           // GP0 Rudder Left
179   #define RUDDER_RIGHT            2           // GP1 Rudder Right
180   #define MOTOR                   4           // GP2 Motor
181   #define ELEVATOR_DOWN           0x10        // GP4 Elevator Down
182   #define ELEVATOR_UP             0x20        // GP5 Elevator Up
183   #define AILERON_LEFT            0           // Not output Aileron Left
184   #define AILERON_RIGHT           0           // Not output Aileron Right
185   #define INVERT_OUTPUT_MASK      0           // No inversions
186   #endif
187
188   //#define RUDDER_RUDDER                     // For auxiliary 4th channel chip
189                                               // Connect pins 2&3 and 6&7 to give double current drive
190   //                  _______
191   //          Vcc -|1     8|- Ground
192   //     Rudder R -|2     7|- Rudder L
193   //     Rudder R -|3     6|- Rudder L
194   //        Input -|4_____5|- Motor
195   //
196   #ifdef RUDDER_RUDDER
197   #define RUDDER_LEFT             3           // GP0,GP1 Rudder Left
198   #define RUDDER_RIGHT            0x30        // GP4,GP5 Rudder Right
199   #define MOTOR                   4           // GP2 Motor
200   #define ELEVATOR_DOWN           0           // Not output Elevator Down
201   #define ELEVATOR_UP             0           // Not output Elevator Up
202   #define AILERON_LEFT            0           // Not output Aileron Left
203   #define AILERON_RIGHT           0           // Not output Aileron Right
204   #define INVERT_OUTPUT_MASK      0           // No inversions
205   #endif
206
207   //#define RFFS_100
208   //                  _______
209   //          Vcc -|1     8|- Ground
210   //        Motor -|2     7|- Aileron
211   //     Elevator -|3     6|- Aileron
212   //        Input -|4_____5|- Elevator
```

```
213  //
214  #ifdef RFFS_100
215  #define ELEVATOR_UP             1           // GP0 Elevator Up
216  #define ELEVATOR DOWN           2           // GP1 Elevator Down
217  #define AILERON_RIGHT           4           // GP2 Aileron Right
218  #define AILERON_LEFT            0x10        // GP4 Aileron Left
219  #define MOTOR                   0x20        // GP5 Motor
220  #define RUDDER LEFT             0           // Not output Rudder Left
221  #define RUDDER_RIGHT            0           // Not output Rudder Right
222  #define INVERT_OUTPUT_MASK      0x1F        // Invert all outputs except Motor
223  #endif
224
225  // Values of control channels
226  signed char count[4];                        // counts for the 4 channels
227  char countPrev[4];                           // previous counts for the 4 channels
228  char GP[4];                                  // GPIO bit for each channel
229  signed char counti;                          // temp count[i]
230  char count2;                                 // Saves motor count[2] in case corrupted while decoding input
231  char motorESC;                               // Mask for motor output to electronic speed control
232
233  // Startup Configuration counters
234  #define CENTER CALIBRATE                    // Enables the center callibration function in StartUp
235  char Param;                                  // Parameter flags saved in EEPROM
236  char MotorEnable;                            // >=0 until throttle has been zero for 1/2 second.  i.e.bit 7 used as a flag
237  char ElevonCounter;                          // >=0 until right stick in corner for 1/2 second while MotorEnable off. bit 7 used as
     a flag
238  char ExpoCounter;                            // >=0 until right stick in corner for 1/2 second while MotorEnable off. bit 7 used as
     a flag
239  #ifdef CENTER CALIBRATE
240  char CenterCounter;                          // >=0 until throttle stick at max for 1/2 second while MotorEnable off. bit 7 used as
     a flag
241  #endif
242  #define ELEVON BIT      1                   // Bit set in Param for Elevon mode
243  #define EXPO_BIT        2                   // Bit set in Param for Expo mode
244  #define CONFIG_COUNTER  28                  // Units are frame times, ~18mS
245
246  char GetOsccal();                            // Defined at end of file
247
248  // Timing parameters
249  // My Tower transmitter:
250  // Frame time: 18,120 uS, constant
251  // Channel time: 6,060 - 9,700 uS
252  // Interframe: 8,420 - 12,060 uS
253  // Ch1(Ail): Left: 1078, Center: 1510, Right: 1930 uS
254  // Ch2(Ele): Down: 1078, Center: 1510,    Up: 1930 uS
255  // Ch3(Thr):  Min: 1048,                    Max: 1940 uS
256  // Ch4(Rud): Left: 1124, Center: 1520, Right: 1940 uS
257  // Ch5(Dum): Cons: 970 uS
258  // Timing pulse width: 300 uS, but signal from Micro Leichty front end is 800-900 nS wide when slightly swamped
259
260  // Use 1500 +- 380 (1120 to 1880) for aileron, elevator and rudder channels
261  // Use 1140 to 1900 for motor channel
262
```

```
263  #define MIN_PULSE_WIDTH     (100/TICK)      // minimum width of timing pulse - units of ticks
264  #define MAX_PULSE_WIDTH     (1000/TICK)     // maximum width of timing pulse - units of ticks
265  #define MIN_PERIOD          (950/TICK)      // minimum channel period - units of ticks
266  #define MAX_PERIOD          (2050/TICK)     // maximum channel period - units of ticks
267  #define CENTER_PERIOD       (1500/TICK)     // default center stick channel period - units of ticks
268  #define MIN_MOTOR_PERIOD    (1140/TICK)     // minimum motor period - units of ticks
269  #define FULL_THROW          (380/TICK)      // aileron & elevator counts at full throw
270
271  #ifdef CENTER_CALIBRATE
272  char Center_Aileron;                        // Measured Aileron at center stick
273  char Center Elevator;                       // Measured Elevator at center stick
274  char Center_Rudder;                         // Measured Rudder at center stick
275  #endif
276  /******************************** Expo ****************************************/
277
278  // Lookup and return value from "exponential" array
279  // Based on (t^2+1)*t/2
280  // Done this way to avoid using data space
281  // No bounds checking - argument must be in range [0..20]
282  // To fit in 8 cycles, must be called as:
283  //      OC(); W = x; y = Expo(); OC();
284  // Do not redefine as char Expo(char x) because that incurs an extra cycle
285  // This procedure needs to be in low memory
286  char Expo() {
287  #asm
288      ADDWF PC,1  ;       jump to appropriate place in data table
289
290      RETLW   .0  // 0
291      RETLW   .0  // 1
292      RETLW   .1  // 2
293      RETLW   .1  // 3
294      RETLW   .2  // 4
295      RETLW   .2  // 5
296      RETLW   .3  // 6
297      RETLW   .3  // 7
298      RETLW   .4  // 8
299      RETLW   .5  // 9
300      RETLW   .6  // 10
301      RETLW   .7  // 11
302      RETLW   .8  // 12
303      RETLW   .9  // 13
304      RETLW   .10 // 14
305      RETLW   .11 // 15
306      RETLW   .12 // 16
307      RETLW   .14 // 17
308      RETLW   .16 // 18
309      RETLW   .18 // 19
310      RETLW   .20 // 20
311  #endasm
312  }
313
314  /******************************** OutputCycle *********************************
315
```

```
316  SaveFSR points to even numbered entries.
317  Buffer size must be even.
318
319  For initialization, can set GPIONext to 0.
320
321  buf:GPIO-0  <- SaveFSR - (value will be copied to GPIOCurr during this entry for output at end)
322      GPIO-1  - (value will be copied to GPIONext ready for next entry)
323      GPIO-2
324      GPIO-3  - value was copied to GPIONext on previous entry for output at beginning
325      GPIO-4  <- SaveFSR - (value will be copied to GPIOCurr during this entry for output at end)
326      GPIO-5  - (value will be copied to GPIONext ready for next entry)
327      GPIO-6
328      . . .
329      GPIO-16
330      GPIO-17 - value was copied to GPIONext on previous entry for output at beginning
331      GPIO-18 <- SaveFSR - (value will be copied to GPIOCurr during this entry for output at end)
332      GPIO-19 - (value will be copied to GPIONext ready for next entry)
333
334  ********************************************************************************/
335
336  // Doubled OutputCycle with lookahead
337  // 30 instructions for 2 outputs
338  // Allows 8 cycles in main for each entry
339  // Total cycle time = 30+8 = 38
340  // Synchronizes exactly to 38 cycles, provided it is called in time
341  // Outputs 2 half cycles to GPIO per entry
342  // 38/2 = 19 => 19*20 = 380 cycles => 2.63KHz-26.3KHz @ 4MHz clock, 5.26KHz-52.6KHz @ 8MHz clock
343  // Also records GPIO in GPIORead halfway through for input timing
344  // Also increments OCTimer once per entry
345  // Touch this procedure at your own risk
346
347  #define TMR0_PERIOD 38      // This is the period of OC() allowing 8 cycles outside
348  #define OutputBuffer_Address 0x60-RESOLUTION            // Carefully chosen so bit 5 signals overflow
349  char OutputBuffer[RESOLUTION] @ OutputBuffer_Address;   // Single GPIO output buffer - same on 12F629 and 635
350  char *SaveFSR;              // Points to next GPIO value to be read
351  char GPIORead;              // Value of GPIO read in OutputCycle2() for external use
352  char OCTimer;               // Incremented once per cycle
353  char GPIOCurr;              // Used internally in OutputCycle2() for value output at end
354  char GPIONext;              // Used internally in OutputCycle2() for value output at beginning of next cycle
355
356  void OC() {
357  #asm
358      // On entry:
359      // GPIONext is next GPIO value to be output
360      // SaveFSR points to value after GPIONext
361      MOVLW .15                       ;   PC += TMR0 & 0xF;   // Synchronize
362      BCF   0x03,RP0                  ;   not strictly needed since #pragma update_RP 0
363      ANDWF TMR0,W                    ;   limit jump to 15
364      ADDWF PC,1                      ;   jump to appropriate place in delay sequence
365
366      NOP  ;  nop();0                     delay sequence to get exact synchronization
367      NOP  ;  nop();1
368      NOP  ;  nop();2
```

```
369       NOP  ;  nop();3
370       NOP  ;  nop();4
371       NOP  ;  nop();5
372       NOP  ;  nop();6     <- set breakpoint here to check for calling in time
373       NOP  ;  nop();7     <- should never be before here
374       NOP  ;  nop();8
375       NOP  ;  nop();9
376       NOP  ;  nop();10
377       NOP  ;  nop();11
378       NOP  ;  nop();12
379       NOP  ;  nop();13
380       NOP  ;  nop();14
381       ; +15 is next instruction
382
383       MOVF  GPIONext,W               ; 1st GPIO output as soon as possible
384       MOVWF GPIO
385
386       MOVLW 5+3-TMR0_PERIOD          ; TMR0 = 5+3-TMR0_PERIOD;   // 5 is value TMR0 should have on getting to next instruction
387                                                                  // 3 is for delay in TMR0 restarting
388       MOVWF TMR0
389
390       MOVF  SaveFSR,W                ; FSR = SaveFSR;
391       MOVWF FSR
392
393   //  MOVF GPIO,W                    ; GPIORead = GPIO;          // Read GPIO for input timing - currently not used
394   //  MOVWF GPIORead                 ; This is half way through testing loop
395
396       INCF  OCTimer                  ; OCTimer++;                // Increment timer for input timing
397
398       MOVF  motorESC,W               ; GPIOCurr = INDF + motorESC;
399       ADDWF INDF,W
400       MOVWF GPIOCurr                 ; will be output at end
401       INCF  FSR,1                    ; FSR++;
402
403       MOVF  motorESC,W               ; GPIONext = INDF + motorESC;
404       ADDWF INDF,W                   ; ready for next cycle
405       MOVWF GPIONext
406       INCF  FSR,1                    ; FSR++;
407
408       MOVLW OutputBuffer Address     ; if (FSR.5) SaveFSR = OutputBuffer_Address; //wraps
409       BTFSS FSR,5                    ; else SaveFSR = FSR
410       MOVF  FSR,W                    ; Wraps when bit 5 set in FSR
411       MOVWF SaveFSR
412
413       MOVF  GPIOCurr,W
414       MOVWF GPIO                     ; 2nd GPIO output, exactly 19 cycles after 1st one
415
416       RETURN
417   #endasm
418   }
419
420   // Initialize OC()
421   void OCInit() {
```

```
422     char i;
423     for (i = 0; i < RESOLUTION; i++) OutputBuffer[i] = INVERT_OUTPUT_MASK;  // Everything off
424     SaveFSR = OutputBuffer;                     // Points to next GPIO value to be read
425     GPIONext = INVERT OUTPUT MASK;              // dummy for first entry
426     TMR0 = -4;                                  // must stay "<0" when first used in OC()
427     OC();
428 }
429
430 /******************************** Chip Init ***************************************/
431
432 // Initialize Chip State
433 void ChipInit() {
434 #pragma update_RP 1                             // Allow setting RP0 for initialization
435
436 #if CLOCK_FREQ == 8
437     IRCF0 = IRCF1 = IRCF2 = 1;                  // Select 8MHz
438     OSCTUNE = GetOsccal();                      // Set specified oscillator calibration
439 #else
440     OSCCAL = GetOsccal();                       // Set built-in oscillator calibration
441 #endif // PIC12F635
442
443     TRISIO = 8;                                 // Input on GP3. 0 = Output, 1 = Input
444     TMR1ON = 0;                                 // TMR1 off
445     TMR1IF = 0;                                 // TMR1 interrupt flag cleared
446     OPTION = 8;                                 // Assign prescaler to WDT, so TMR0 clock is 1:1
447     T1CKPS0 = 1; T1CKPS1 = 1;                   // 8:1 prescale on TMR1
448     TMR1ON = 1;                                 // TMR1 on - never turned off again
449
450     CM2 = 1; CM1 = 1; CM0 = 1;                  // Digital I/O
451 #ifdef HAS_AD
452     ANSEL = 0;                                  //Turn off ADC on the 12F675
453 #endif
454     GPIO = INVERT_OUTPUT_MASK;                  // Zero coding all outputs
455
456 #pragma update_RP 0                             // Don't set RP0 any more - needed to be able to call OC() fast enough
457
458 }
459
460 /******************************** Jingle ******************************************/
461
462 #define on ((AILERON_LEFT + ELEVATOR_DOWN + RUDDER_LEFT) ^ INVERT_OUTPUT_MASK)
463 #define off INVERT_OUTPUT_MASK
464 #define PER (8/CLOCK_FREQ)
465 #define BEAT 200/PER
466 // Re-use GP[0] for variables
467 #define lastt GP[0]             // last value read from timer for counting ms
468 #define ms GP[1]                // millisecond timer
469 #define endt GP[2]              // time sought
470 #define t GP[3]                 // current time
471
472 // Wait n * 4 uS
473 // Increment ms every 1 mS
474 void Wait (char n) {
```

```
475
476         t = TMR1L;
477         if (t < lastt) ms++;
478         lastt = t;
479         endt = t + n;  //let it overflow
480         while (TMR1L != endt);
481     }
482
483     // Play a note for one beat (half period in 4mS units)
484     void Note(char tone) {
485         for (ms = 0; ms < BEAT;) {
486             GPIO = on; Wait(tone);
487             GPIO = 0; Wait(tone);
488         }
489     }
490
491     // Play nothing for one beat
492     void Gap() {
493         for (ms = 0; ms < BEAT;) Wait(10);
494
495     }
496
497     // Play a jingle
498     void Jingle() {
499         lastt = 0;
500         ms = 0;
501         TMR1L = 0;
502         Note(174/PER);
503         Note(207/PER);
504         Gap();
505         Gap();
506         Gap();
507         Note(207/PER);
508         Note(195/PER);
509         Note(174/PER);
510         Note(103/PER);
511         Gap();
512         Note(103/PER);
513         Gap();
514         Note(130/PER);
515         Gap();
516         Gap();
517         Gap();
518     }
519
520     /******************************** EEPROM Read and Write ***************************************/
521
522     #define MAGIC 42                                    // In word 0 of EEPROM says it is initialized
523
524     // Write data to the EEPROM
525     // Beware - it takes about 4 mS for the write to complete
526     #pragma update_RP 1                                 // Allow setting RP0 for initialization
527     void WriteEEPROM(char addr, char data) {
```

```
528         EEIF = 0;                                   // clear done flag
529         EEADR = addr;                               // set up address
530     #if CLOCK_FREQ == 8
531         EEDAT = data;                               // set up data to write
532     #else
533         EEDATA = data;                              // set up data to write
534     #endif
535         WREN = 1;                                   // enable writing
536         EECON2 = 0x55;                              // required sequence
537         EECON2 = 0xAA;
538         WR = 1;                                     // do the write
539         WREN = 0;                                   // disable other writes
540         while (EEIF == 0) {};                       // wait for write to complete - about 4mS
541         EEIF = 0;                                   // clear done flag
542     #pragma update RP 0                             // Don't set RP0 any more - needed to be able to call OC() fast enough
543         RP0 = 0;                                    // assumed in main()
544     #if CLOCK_FREQ == 8
545         RP1 = 0;                                    // assumed in main()
546     #endif
547     }
548
549     #pragma update RP 1                             // Allow setting RP0 for initialization
550     char ReadEEPROM(char addr) {
551         char dat;
552         EEADR = addr;
553         RD = 1;                                     // do the read
554     #if CLOCK_FREQ == 8
555         dat = EEDAT;
556     #else
557         dat = EEDATA;
558     #endif
559     #pragma update_RP 0                             // Don't set RP0 any more - needed to be able to call OC() fast enough
560         RP0 = 0;
561     #if CLOCK_FREQ == 8
562         RP1 = 0;                                    // assumed in main()
563     #endif
564         return dat;                                 // result immediately available
565     }
566
567     void ParamInit() {
568         // Get parameters from EEPROM, if set
569         // EEPROM word zero must be 42
570         // EEPROM word 1 stores Param
571         Param = ReadEEPROM(0);
572         if (Param == MAGIC) {                       // 42 used as keyword to say EEPROM has been written
573             Param = ReadEEPROM(1);
574         } else {
575             WriteEEPROM(0, MAGIC);                  // Do this here because it takes 4mS to settle
576             Param = 0;                              // initialize all parameter flags to 0
577         }
578     }
579
580     /******************************** StartupConfig ***************************************/
```

```
581
582 // Set various parameters while in Startup (before MotorEnable)
583 // Right stick Right-Top    (+,-): Elevon mode          Elevon Off ---- Elevon On
584 // Right stick Left-Top     (-,-): Aileron-Elevator       |               |
585 // Right stick Right-Bottom (-,+): Expo                  |               |
586 // Right stick Left-Bottom  (+,+): Linear              Expo Off ------- Expo on
587 // Throttle at max - Center Calibrate Aileron, Elevator and Rudder
588
589 void StartupInit() {
590         // Initialize Startup Configuration counters
591     MotorEnable = CONFIG_COUNTER;
592     ElevonCounter = CONFIG_COUNTER;
593     ExpoCounter = CONFIG_COUNTER;
594 #ifdef CENTER_CALIBRATE
595     CenterCounter = CONFIG_COUNTER;
596
597     Center_Aileron = CENTER_PERIOD;
598     Center_Elevator = CENTER_PERIOD;
599     Center_Rudder = CENTER_PERIOD;
600 #endif
601 }
602
603 void StartupConfig() {
604
605     if (MotorEnable.7 == 0) {                       OC();
606
607         // Don't start motor output until throttle stick has stayed at zero for 1/2 second
608         if (count[2]) MotorEnable = CONFIG_COUNTER;         // Didn't stay zero long enough, start again
609         MotorEnable--;                              OC();   // decrement towards -1
610         if (MotorEnable.7) {
611             // Here exactly once, when MotorEnable first goes negative
612             // Record parameters in EEPROM
613                                                             // Should be easily done by the time we
614                                                             //  get here since StartupConfig() called about every 18mS
615             WriteEEPROM(1, Param);                  OC();   // Will get one glitch in output, but everything should be zero anyway
616             count[0] = RESOLUTION/2;                        // Momentarily move surfaces as signal that mode is set
617             count[1] = RESOLUTION/2;
618         }
619
620         // Check for Elevon mode - Full Down, Left-Right for Off-On
621         if (ElevonCounter.7 == 0) {                 OC();
622             if (count[1] > -FULL_THROW) ElevonCounter = CONFIG_COUNTER; OC(); // Didn't stay at top long enough, start again
623             if (count[0] < FULL_THROW)  {           OC();   // Check full left or full right
624                 if (count[0] > -FULL_THROW) ElevonCounter = CONFIG_COUNTER;  // Didn't stay extreme, start again
625             }                                       OC();
626             ElevonCounter--;                                // decrement towards -1
627         } else {                                    OC();
628             // Here whenever ElevonCounter makes it to -1
629             if (count[0] >= FULL_THROW) {
630                 Param |= ELEVON_BIT;                OC();   // Set elevon mode on
631             } else Param &= ~ELEVON_BIT;             OC();   // Set elevon mode off
632             ElevonCounter = CONFIG_COUNTER;                 // Reset counter
633             count[0] = RESOLUTION/2;                        // Momentarily move surfaces as signal that mode is set
```

```
634             count[1] = RESOLUTION/2;
635         }                                           OC();
636
637         // Check for Expo mode - Full Up, Left-Right for Off-On
638         if (ExpoCounter.7 == 0) {                   OC();
639             if (count[1] < FULL_THROW-1) ExpoCounter = CONFIG_COUNTER; OC(); // Didn't stay at top long enough, start again
640             if (count[0] < FULL_THROW-1) {          OC();   // Check full left or full right
641                 if (count[0] > -FULL THROW-1) ExpoCounter = CONFIG_COUNTER;  // Didn't stay extreme, start again
642             }                                       OC();
643             ExpoCounter--;                                  // decrement towards -1
644         } else {                                    OC();
645             // Here whenever ExpoCounter makes it to -1
646             if (count[0] >= FULL_THROW-1) {
647                 Param |= EXPO_BIT;                  OC();   // Set Expo mode on
648             } else Param &= ~EXPO BIT;              OC();   // Set Expo mode off
649             ExpoCounter = CONFIG_COUNTER;                   // Reset counter
650             count[0] = 0;                       // Momentarily move surfaces as signal that mode is set
651             count[1] = 0;
652         }                                           OC();
653
654 #ifdef CENTER_CALIBRATE
655         // Check for Center Calibrate - Throttle full On
656         if (CenterCounter.7 == 0) {                 OC();
657             if (count[2] < FULL_THROW+FULL_THROW) CenterCounter = CONFIG_COUNTER; OC(); // Didn't stay at top long enough, start
    again
658             CenterCounter--;                                // decrement towards -1
659         } else {                                    OC();
660             // Here whenever CenterCounter makes it to -1
661             Center_Aileron += count[0];
662             Center_Elevator += count[1];
663             Center_Rudder += count[3];
664             CenterCounter = CONFIG_COUNTER;                 // Reset counter
665             count[0] = RESOLUTION/2;                        // Momentarily move surfaces as signal that mode is set
666             count[1] = RESOLUTION/2;
667         }                                           OC();
668 #endif // CENTER_CALIBRATE
669
670         count[2] = 0;                                   // Zero throttle anyway
671     }                                               OC();
672 }
673
674 /******************************** SynchUp ****************************************/
675
676 // Seeks INPUT_ON
677 // Returns 0 if OK, else 1
678 // Fail if wait more than 14.442mS
679 // If successful, OCTimer will have been set to zero at the transition
680 // Also OCHalfTimer will be 1 if transition found earlier during OutputCycle else 0
681 // Timing of synch transition is critical - need to examine assembler code
682 // Important to have only 1 call to OC() for each test of GPIO.3
683 // Note - value of input signal corresponding to INPUT_ON and INPUT_OFF depends on value of pulse
684
685 char OCHalfTimer;
```

```
686
687 char SynchUp() {
688                                                 OC();
689     OCHalfTimer = 0;
690     OCTimer = 0;
691     do {                                        OC();
692         if (INPUT_ON) goto found;
693     } while (OCTimer.7 == 0);
694     // here if GPIO still Off when OCTimer reaches 128, in 4.814mS (2.407 for 8MHz)
695     OCTimer = 0;
696     do {                                        OC();
697         if (INPUT_ON) goto found;
698     } while (OCTimer.7 == 0);
699     // here if GPIO still Off when OCTimer reaches 128, in 4.814mS (2.407 for 8MHz)
700     OCTimer = 0;
701     do {                                        OC();
702         if (INPUT_ON) goto found;
703     } while (OCTimer.7 == 0);
704     // here if GPIO still Off when OCTimer reaches 128, in 4.814mS (2.407 for 8MHz)
705
706 #if CLOCK_FREQ == 8
707     OCTimer = 0;
708     do {                                        OC();
709         if (INPUT_ON) goto found;
710     } while (OCTimer.7 == 0);
711     // here if GPIO still Off when OCTimer reaches 128, in 4.814mS (2.407 for 8MHz)
712     OCTimer = 0;
713     do {                                        OC();
714         if (INPUT ON) goto found;
715     } while (OCTimer.7 == 0);
716     // here if GPIO still Off when OCTimer reaches 128, in 4.814mS (2.407 for 8MHz)
717     OCTimer = 0;
718     do {                                        OC();
719         if (INPUT_ON) goto found;
720     } while (OCTimer.7 == 0);
721     // here if GPIO still Off when OCTimer reaches 128, in 4.814mS (2.407 for 8MHz)
722 #endif // CLOCK_FREQ == 8
723
724     // Timed out
725                                                 OC();
726     return 1;
727
728 found:
729     OCTimer = 0;                                OC();
730     if ((GPIORead & INPUT_PIN) == pulse) OCHalfTimer++; OC();
731     return 0;
732 }
733
734
735 /****************************** DecodePulses ****************************************/
736
737 // Decode one set of input timing pulses
738 // return 1 if fails, else 0
```

```
739  // Assumes OCTimer was re-started at beginning of first pulse
740  // Note - value of input signal corresponding to INPUT_ON and INPUT_OFF depends on value of pulse
741
742  char DecodePulses() {                           OC();
743      char i,j;
744
745      for (i = 0; i < 4; i++) {                   OC();   // for each channel
746
747          // Check pulse remains for at least MIN_PULSE_WIDTH
748          OCTimer -= MAX_PULSE_WIDTH;             // as if set to -MAX_PULSE_WIDTH at beginning of period
749          do {                                    OC();
750              if (INPUT_OFF) goto endPulseFound;
751          } while (OCTimer.7);                    OC();
752
753          // Here if timed out
754          return 1;
755
756  endPulseFound:                                  OC();
757          // End of pulse found, now wait for beginning of next pulse.  Timing critical
758          counti = OCTimer + MAX_PULSE_WIDTH;     OC();
759
760          // Check if pulse was long enough
761          if (counti < MIN_PULSE_WIDTH) return 1;         // Can't be since so many calls to OC() before first test
762
763          OCTimer += MAX_PULSE_WIDTH - MAX_PERIOD;        // as if set to -MAX_PERIOD at beginning of period
764          // Must have only one call to OC() in following loop
765          do {                                    OC();
766              if (INPUT_ON) goto endPeriodFound;
767          } while (OCTimer.7);                    OC();
768
769          // Here if timed out
770          return 1;
771
772  endPeriodFound:                                 OC();   //+1
773          // Capture timing first
774          counti = OCTimer + MAX_PERIOD -1;       OC();   // -1 because of OC() since transition
775          count[i] = counti;
776          OCTimer = 2;                            OC();   // now counting from this transition
777                                                          // 2 because of 2 OC()s since transition
778
779          // Check if period was long enough
780          if (counti < MIN_PERIOD) return 1;
781                                                  OC();
782      }       // for (i = 0; i < 4; i++) {    // for each channel
783                                                  OC();
784      // Succeeded
785      // Shuffle to get motor channel in count[2]
786  #define swap(a,b) counti=count[a];OC();count[a]=count[b];count[b]=counti
787
788      if (MotorChannel == 3) {                            // swap with count[2]
789          swap(2,3);                              OC();
790          return 0;
791      }                                           OC();
```

```
792         if (MotorChannel == 1) {                            // swap with count[2]
793             swap(1,2);                                  OC();
794             return 0;
795         }                                               OC();
796         if (MotorChannel == 0) {                            // cycle count[0..2]
797             swap(0,1);
798             swap(1,2);
799         }                                               OC();
800         return 0;
801     }
802
803     /******************************** ProcessCounts ****************************************/
804
805     // Process raw count[] array into output values
806     void ProcessCounts() {                                              OC();
807         char i;
808
809         // Remove expected 1 LSB jitter due to asynchronous sampling by averaging with previous counts
810         count[0] = countPrev[0] = (count[0] + countPrev[0]) >> 1;    OC();
811         count[1] = countPrev[1] = (count[1] + countPrev[1]) >> 1;    OC();
812         count[2] = countPrev[2] = (count[2] + countPrev[2]) >> 1;    OC();
813         count[3] = countPrev[3] = (count[3] + countPrev[3]) >> 1;    OC();
814
815     #ifdef MOTOR_PWM
816     // Cause OC() to output a PWM motor pulse
817     // Done here to get smoothed count
818     // Note, spacing between pulses is timed by input frame rate
819         motorESC = MOTOR;
820         for (i = count[2]; i; i--) OC();
821         motorESC = 0;                                   OC();
822     #endif
823
824         // Normalize counts
825     #ifdef CENTER_CALIBRATE
826         count[0] -= Center_Aileron;
827         count[1] -= Center_Elevator;
828         count[2] -= MIN_MOTOR_PERIOD;
829         count[3] -= Center_Rudder;                      OC();
830     #else
831         count[0] -= CENTER_PERIOD;
832         count[1] -= CENTER_PERIOD;
833         count[2] -= MIN_MOTOR_PERIOD;
834         count[3] -= CENTER_PERIOD;                      OC();
835     #endif
836
837         // Motor
838         if (count[2].7) count[2] = 0;                       // Clip to 0 before StartupConfig
839
840         // Check Startup Configurations
841         // Done here because we want actual stick positions
842         StartupConfig();                                OC();
843
844         // Elevon mixing - here because need signed, unscaled quantities
```

```
845         if (Param & ELEVON_BIT) {
846             counti = count[0];
847             count[0] -= count[1];
848             count[1] += counti;
849         }                                           OC();
850
851         // Map count[] values in place and set GPIO bits
852         // Aileron
853         if (count[0].7) {                                   // i.e. if negative
854             count[0] = -count[0];
855             GP[0] = AILERON_LEFT;
856         } else {
857             GP[0] = AILERON_RIGHT;
858         }                                           OC();
859
860         // Elevator
861         if (count[1].7) {                                   // i.e. if negative
862             count[1] = -count[1];
863             GP[1] = ELEVATOR_DOWN;
864         } else {
865             GP[1] = ELEVATOR_UP;
866         }                                           OC();
867
868         // Channel 3
869         if (count[3].7) {                                   // i.e. if negative
870             count[3] = -count[3];
871             GP[3] = RUDDER_LEFT;
872         } else {
873             GP[3] = RUDDER RIGHT;
874         }                                           OC();
875
876     #if CLOCK_FREQ == 4
877         count[0] += count[0];                               // Double value to get range [0..20]
878         count[1] += count[1];
879         //count[2] += count[2];                             // Motor already full resolution
880         count[3] += count[3];                       OC();
881     #else // for CLOCK_FREQ == 8
882         count[2] = count[2] >> 1;                   OC();   // Motor has double needed resolution
883     #endif
884
885         // Expo must go here - count[]s are positive but not yet scaled
886         if (count[0] > RESOLUTION) count[0] = RESOLUTION;   OC();
887         if (count[1] > RESOLUTION) count[1] = RESOLUTION;   OC();
888         if (count[2] > RESOLUTION) count[2] = RESOLUTION;   OC();
889         if (count[3] > RESOLUTION) count[3] = RESOLUTION;   OC();
890
891         if (Param & EXPO BIT) {                     OC();
892             W = count[0]; count[0] = Expo();        OC();
893             W = count[1]; count[1] = Expo();        OC();
894             // count[2], motor, left linear
895             W = count[3]; count[3] = Expo();
896         }                                           OC();
897
```

```
898         // Scale - no scaling needed
899     }
900
901     /****************************** BuildGPIOArray ****************************************/
902
903     // Build the GPIO array
904     // Use double buffering if appropriate and enough space
905     // Expects count[] and GP[] to be set up
906     // For inverted outputs enable the following statement
907     //#define INVERT_OUTPUT
908     // or rely on
909
910     void BuildGPIOArray() {
911         char GPVal;                                     // Build GPIO value
912         char i,j;                                       // Loop indices
913         signed char y[4];                               // Accumulators in PFM
914     //#define PWM
915     #ifdef PWM  // Pulse Width Modulation output - needs double buffering
916         // No need to sort, just cycle through each entry setting bits
917         for (i = 0; i < RESOLUTION; i++) {          OC();
918             GPVal = 0;
919             for (j = 0; j < 4; j++) {               OC();   // For each channel
920                 if (count[j] > i) {                 OC();
921                     GPVal += GP[j];
922                 }                                   OC();
923             }                                       OC();
924             buf[i] = GPVal;                         OC();
925         }
926
927     #else   //PFM Pulse Frequency Modulation output - overwrite buffer in place
928
929         y[0] = 0; y[1] = 0; y[2] = 0; y[3] = 0;     OC();
930         for (i = 0; i < RESOLUTION; i++) {          OC();
931             GPVal = 0;
932
933             // loop unwound for speed (far fewer OC()) (it's also less code)
934             y[0] -= count[0];                               // Channel 0
935             if (y[0] < 0) {                         OC();
936                 y[0] += RESOLUTION;
937                 GPVal += GP[0];
938             }                                       OC();
939
940             y[1] -= count[1];                               // Channel 1
941             if (y[1] < 0) {                         OC();
942                 y[1] += RESOLUTION;
943                 GPVal += GP[1];
944             }                                       OC();
945
946     #ifndef MOTOR_PWM
947             // Include normal throttle pulsed output
948             y[2] -= count[2];                               // Channel 2
949             if (y[2] < 0) {                         OC();
950                 y[2] += RESOLUTION;
```

```
951             GPVal += GP[2];
952         }                                       OC();
953 #endif
954
955         y[3] -= count[3];                               // Channel 3
956         if (y[3] < 0) {                         OC();
957             y[3] += RESOLUTION;
958             GPVal += GP[3];
959         }                                       OC();
960
961         OutputBuffer[i] = GPVal ^ INVERT_OUTPUT_MASK;   OC();   // Invert required outputs
962     }
963                                                 OC();
964 #endif //PWM
965
966 }
967
968 /******************************** StandBy ***************************************/
969
970 // Progressively shut down controls
971 // After 1.5 secs: Aileron to 1/5 left, Elevator to neutral, Motor unchanged
972 // After 3.0 secs: Aileron at 1/5 left, Elevator at neutral, Motor reduced by 1 unit or zero
973 // Every 1.5 secs: ditto
974
975 #define STANDBY_PERIOD (6*CLOCK_FREQ/4)                     // ~1.5 sec
976 char StandByCount;
977
978 #define STANDBY_CYCLES 4                                    // Number of complete correct consecutive frames needed to assume good
    signal
979 char StandByTMR1H;
980
981 #define TMR1_REACHED(T) (((TMR1H - T) & 0x80) == 0)
982 #define TMR1_NOT_REACHED(T) (((TMR1H - T) & 0x80) != 0)
983
984 void StandByInit() {
985     // Initialize motor for StandBy() during initial synching
986     count[2] = 0;
987     count2 = 0;
988     GP[2] = MOTOR;
989 }
990
991 void StandByTurnOn() {                              OC();   // Turn on StandBy
992     StandByCount = STANDBY_PERIOD;
993     StandByTMR1H = TMR1H + 128;
994 }
995
996 void StandBy() {                                    OC();
997     if (TMR1_REACHED(StandByTMR1H)) {               OC();
998         StandByTMR1H = StandByTMR1H + 128;                  // another half period of TMR1H (262 mS or 131 mS)
999         if (StandByCount == 0) {                    OC();
1000            StandByCount = STANDBY_PERIOD;
1001
1002            count[0] = FULL_THROW/5;
```

```
1003            GP[0] = AILERON_LEFT;                   OC();   // Aileron 1/5 left, or Left Elevon
1004
1005            if (Param & ELEVON_BIT) {
1006                count[1] = FULL_THROW/5;                    // Right Elevon 1/5 down
1007            } else {
1008                count[1] = 0;                               // Elevator neutral
1009            }                                       OC();
1010            GP[1] = ELEVATOR_DOWN;
1011
1012            count[2] = count2;                              // motor
1013
1014            count[3] = FULL_THROW/5;                        // Rudder 1/5 left
1015            GP[3] = RUDDER_LEFT;                    OC();
1016
1017            BuildGPIOArray();                       OC();
1018
1019            counti = count[2] >> 1;                         // multiply motor speed by 3/4
1020            counti = count[2] + counti;             OC();
1021            count[2] = counti >> 1;                 OC();
1022            count2 = count[2];                              // save motor count because count[2] may get random number in a
    corrupt frame
1023        }
1024        StandByCount--;
1025    }
1026                                                    OC();
1027}
1028
1029/******************************** FindMinInterFrame *****************************/
1030
1031// Seek period of no transitions for 4mS
1032// Returns 0 if successful, 1 if not found in time
1033// Sets 'pulse' depending on positive- or negative-going pulses
1034// Calls Standby()
1035// Note - value of input signal corresponding to INPUT_ON and INPUT_OFF depends on value of pulse
1036
1037#define MIN INTERFRAME TIME (2*CLOCK FREQ/4)                // ~4 mS
1038char MinInterFrameTMR1H;                                    // Times 4 mS minimum gap
1039
1040#define BAD_INTERFRAME_TIME (6*CLOCK_FREQ/4)                // ~12mS Ticks of TMR1H after which declare bad interframe - 6mS
1041char BadInterFrameTMR1H;                                    // Times too long until start of interframe gap found
1042
1043char FindMinInterFrame() {                                  OC();
1044
1045    BadInterFrameTMR1H = TMR1H + BAD_INTERFRAME_TIME;           // Time when waited too long
1046    MinInterFrameTMR1H = TMR1H + MIN_INTERFRAME_TIME;       OC();
1047
1048    while(TMR1 NOT REACHED(MinInterFrameTMR1H)) {           OC();
1049        if (INPUT_ON) {                                     OC();   // went On too soon
1050            MinInterFrameTMR1H = TMR1H + MIN_INTERFRAME_TIME;       // Reset timer.
1051            pulse = INPUT_PIN - pulse;                      OC();   // toggle pulse and try again
1052            // with redefined pulse, input just went Off
1053            if (TMR1_REACHED(BadInterFrameTMR1H)) {         OC();   // Failed - been too long to start again looking for gap
1054                return 1;
```

```
1055            }
1056        }                                                   OC();
1057    }
1058                                                            OC();
1059    // Here if stayed off long enough - gap found
1060    return 0;
1061
1062 }
1063
1064 /******************************** SetMotorChannel ****************************/
1065
1066 // Set MotorChannel to be the channel furthest from CENTER_PERIOD
1067 // Other channels are assumed in the order Aileron, Elevator, Rudder
1068 #define max GP[0]                                          // Borrow GP[0]
1069 void SetMotorChannel() {                                   OC();
1070     char i;
1071
1072     MotorChannel = 2;
1073
1074     // Loop until got a good frame
1075     while (1) {                                            OC();
1076         while (FindMinInterFrame()) OC();
1077
1078         // input stayed Off long enough - try to Synch
1079         if (SynchUp()) continue;                           OC();
1080
1081         // Here when found first pulse transition after inter-frame period
1082         if (DecodePulses() == 0) break;                    OC();
1083     }
1084
1085     // Find channel furthest from midpoint
1086     max = 0;                                               OC();
1087     for (i = 0; i < 4; i++) {                              OC();
1088         counti = count[i];                                 OC();
1089         counti -= CENTER_PERIOD;
1090         if (counti < 0) counti = -counti;                  OC();
1091         if (counti >= max) {
1092             max = counti;
1093             MotorChannel = i;
1094         }                                                  OC();
1095     }                                                      OC();
1096 }
1097
1098 /*****************************************************************************/
1099 /******************************** MAIN ***************************************/
1100 /*****************************************************************************/
1101
1102 void main()
1103 {
1104     char onStandBy;                                // non-zero if need to call StandBy
1105     char i;
1106
1107     ChipInit();
```

```
1108  #ifndef SIMULATOR
1109      Jingle();
1110  #endif
1111      ParamInit();
1112      StandByInit();
1113      onStandBy = 0;
1114      StartupInit();
1115      pulse = 0;                                      // sense of input timing pulses - confirmed or changed during Startup
1116                                                      // Takes on values [0,INPUT_PIN]
1117      // Initialize countPrev[]
1118      for (i = 0; i < 4; i++) countPrev[i] = CENTER_PERIOD;   // Throttle in center for Startup Configuration
1119      motorESC = 0;
1120
1121      OCInit();
1122
1123      SetMotorChannel();
1124
1125      // Loop for a restart
1126      while (1) {
1127
1128          if (onStandBy == 0) StandByTurnOn();            OC();
1129          onStandBy = STANDBY_CYCLES;                     OC();
1130
1131          // Loop while staying in synch
1132          while (1) {                                     OC();
1133  //          motorESC = MOTOR; OC(); motorESC = 0;       OC();   //Pulse on motor pin for oscilloscope
1134              if (onStandBy) StandBy();                   OC();
1135              if (FindMinInterFrame()) break;             OC();
1136
1137              // input stayed Off long enough - try to Synch
1138              if (SynchUp()) break;                       OC();
1139
1140              // Here when found first pulse transition after inter-frame period
1141              if (DecodePulses()) break;                  OC();
1142
1143              // Raw periods are now in count[]
1144              ProcessCounts();                            OC();
1145
1146              // Processed control data now in count[]
1147              if (onStandBy) {
1148                  onStandBy--;                            OC();
1149              } else {                                    OC();
1150                  BuildGPIOArray();                       OC();
1151                  count2 = count[2];                              // save good motor count in case corrupted in DecodePulses() -
      used in Standby()
1152              }
1153          }                                               OC();
1154      }
1155  } // main
1156
1157  #ifdef PIC12F683
1158  #pragma origin 0x7FF
1159  #else
```

```
1160  #pragma origin 0x3FF
1161  #endif
1162  // This is to enable us to create a call to the oscillator calibration value in 12F629
1163  // On 12F629 this is a dummy - actual procedure is preset in chip and will override
1164  // On 12F635 this procedure runs and returns value shown - need to calibrate manually
1165  // A change of 1 causes about 1% change in frequency
1166  // No need to recompile to change value - can set the value in WinPic's buffer and rewrite the PIC
1167  char GetOsccal() {return 0x0;}
1168
1169  // Initialize EEPROM data
1170  #define EEPROM_START 0x2100
1171  #pragma cdata[EEPROM_START]
1172  #pragma cdata[] = MAGIC, 0x00                          // AILERON-ELEVATOR mode, EXPO off
1173
```