```
/* Introduction for remote control module:
  This module provide interface to work with remote control receiver signals

    1. init_remote_control()
    This function initial all the variables for remote control signal processing.
    It must be called in initialization stage

    2. HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
    This function process the input capture channels interrupt to calculate the
    pulse of each RC channel and save the pulse width value into global variable
    rc_t[4]. The drfault time step is 0.25us/LSB.
    It also convert the pulse width signal into real control signal stored in
    global variable: gAIL, gELE, gTHR, gRUD (by calling update_rc_data(idx))
    gAIL, gELE and gRUD are 0 centerred (+/-0.5ms with 0.25us/LSB). gTHE is 0~1ms
    with 0.25us/LSB.

    3. HAL_SYSTICK_Callback()
    This function calls the user timer processing function first. And then it will
    check if RC signal is time out or not. A time out flag will be set (rc_connection_flag)
    if no RC pulse is received within certain time

  Note
    If AUTO_CONNECTION_CENTER is set to 1, the first 4 sets of RC data during
    connection, will be averaged and used as the center of Roll, Pitch and Yaw
    control.
*/

#include "rc.h"

#define AUTO_CONNECTION_CENTER  0

const float max_pitch_rad = PI*PITCH_MAX_DEG/180.0;
const float max_roll_rad = PI*ROLL_MAX_DEG/180.0;
const float max_yaw_rad = PI*YAW_MAX_DEG/180.0;
int32_t t1;


int32_t ail_center = AIL_MIDDLE;
int32_t ele_center = ELE_MIDDLE;
int32_t rud_center = RUD_MIDDLE;

int32_t rc_cnt = 0;
int32_t rc_acc[4] = {0};

int32_t rc_z_control_flag = 1;

extern int32_t rc_cal_flag;
extern int32_t rc_enable_motor;
extern int32_t fly_ready;

```

```
GPIO_TypeDef* RC_Channel_Ports[4] =
    {
      RC_CHANNEL1_PORT,
      RC_CHANNEL2_PORT,
      RC_CHANNEL3_PORT,
      RC_CHANNEL4_PORT
    };

uint16_t RC_Channel_Pins[4] =
    {
      RC_CHANNEL1_PIN,
      RC_CHANNEL2_PIN,
      RC_CHANNEL3_PIN,
      RC_CHANNEL4_PIN
    };

volatile int rc_timeout;    // R/C timeout counter
char rc_connection_flag;    // R/C connection status
char rc_flag[4];            // flag for received input capture interrupt count
/* timer data for rising and falling edge and pulse width */
int rc_t_rise[4], rc_t_fall[4], rc_t[4];
/* Global R/C data */
int16_t gAIL, gELE, gTHR, gRUD;


// A queue for testing purpose (to print R/C data in main function)
Queue_TypeDef que;
int32_t cnt;

// privite function
void init_rc_variables(void);

void init_remote_control(void)
{
  rc_connection_flag = 0;
  rc_timeout = 1000;

  // Initial R/C global variables
  gAIL = 0;
  gELE = 0;
  gTHR = 0;
  gRUD = 0;

  init_rc_variables();
  // queue for test purpose
  cnt = 0;
  init_queue(&que);
}

void init_rc_variables(void)
```

```
101 {
102   uint32_t i;
103   rc_connection_flag = 0;
104   for (i=0;i<4;i++)
105   {
106     rc_flag[i] = 0;
107     rc_t_rise[i] = 0;
108     rc_t_fall[i] = 0;
109     rc_t[i] = 0;
110   }
111 }
112
113 void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
114 {
115   int32_t timcnt, idx;
116
117   // save the counter data
118   switch (htim->Channel)
119   {
120     case HAL_TIM_ACTIVE_CHANNEL_1: idx = 0; timcnt = htim->Instance->CCR1; break;
121     case HAL_TIM_ACTIVE_CHANNEL_2: idx = 1; timcnt = htim->Instance->CCR2; break;
122     case HAL_TIM_ACTIVE_CHANNEL_3: idx = 2; timcnt = htim->Instance->CCR3; break;
123     case HAL_TIM_ACTIVE_CHANNEL_4: idx = 3; timcnt = htim->Instance->CCR4; break;
124     default: idx = -1;
125   }
126
127   if (idx != -1)
128   {
129     rc_timeout = 0;     // Reset timeout count
130     if (rc_flag[idx] < 2)
131       rc_flag[idx]++;
132     if (HAL_GPIO_ReadPin(RC_Channel_Ports[idx], RC_Channel_Pins[idx]) == GPIO_PIN_SET)
133     {   // rising edge
134       rc_t_rise[idx] = timcnt;
135     }
136     else
137     {   // falling edge
138       rc_t_fall[idx] = timcnt;
139       if (rc_flag[idx] >= 2)  // be sure to calculate signal width after at least a pair
    of interrupt
140       {
141         if (rc_t_fall[idx] > rc_t_rise[idx])
142           rc_t[idx] = rc_t_fall[idx] - rc_t_rise[idx];
143         else
144           rc_t[idx] = rc_t_fall[idx] - rc_t_rise[idx] + 32768;
145       }
146       update_rc_data(idx);
147
148       if (AUTO_CONNECTION_CENTER)
149       {
```

マイコンのインプットキャプチャ機能を用いてラジコン受信器から受けたPWM信号のパルス幅を計測する。

受信機からの入力信号線でパルスエッジを検出した場合はidxに0〜3のチャネル番号を代入し、そうでなければidxに-1を代入する。

立ち上がりエッジを検出した場合は、現在のタイマカウンタ値を変数 rc_t_rise に保存する。

立ち下がりエッジを検出した場合は、現在のタイマカウンタ値から立ち上がりエッジ検出時のタイマカウンタ値を引いてパルス幅を計算し、関数update_rc_data()を呼び出す。

```
150           // Automatic use average of 4 RC data during connecting as the center of RC
151           //if (rc_cnt < 16)
152           if (rc_cnt < 4)
153           {
154               rc_acc[idx] += rc_t[idx];
155               rc_cnt++;
156               //if (rc_cnt == 16)
157               if (rc_cnt == 4)
158               {
159                   ail_center = rc_acc[0] / 4;
160                   ele_center = rc_acc[1] / 4;
161                   rud_center = rc_acc[3] / 4;
162               }
163           }
164       }
165
166       // For debug purpose
167       // Add channel & width info into a queue for printing
168       add_queue(&que, idx, rc_t[idx]);
169     }
170   }
171 }
172
173
174 void HAL_SYSTICK_Callback(void)
175 {
176   // Process user timer
177   User_Timer_Callback();
178   // Count rc_timeout up to 1s
179   if (rc_timeout < 1000)
180     rc_timeout++;
181   if (rc_timeout > RC_TIMEOUT_VALUE)
182     init_rc_variables();
183   rc_connection_flag = (rc_timeout <= RC_TIMEOUT_VALUE);
184 }
185
186
187 /* Update global variables of R/C data */
188 void update_rc_data(int32_t idx)
189 {
190   switch (idx)
191   {
192     case 0: gAIL = rc_t[0] - ail_center; break;
193     case 1: gELE = rc_t[1] - ele_center; break;
194     case 2: gTHR = (rc_t[2] > THR_BOTTOM) ? (rc_t[2] - THR_BOTTOM) : 0; break;
195     case 3: gRUD = rc_t[3] - rud_center; break;
196     default: break;
197   }
198
199   if ( (gTHR == 0) && (gELE < - RC_CAL_THRESHOLD) && (gAIL > RC_CAL_THRESHOLD) && (gRUD <
```

システムクロック割り込み処理関数：
1ms 毎に呼び出される。

timer.c 内にて定義

一定時間（RC_TIMEOUT_VALUE）以上受信機からの信号が途絶えた場合、信号入力なしとみなし、変数初期化を行う。

gAIL, gELE, gTHR, gRUD に操作量[LSB]を代入。

```
- RC_CAL_THRESHOLD))
200   {
201     rc_cal_flag = 1;
202   }
203
204   if ( (gTHR == 0) && (gELE < - RC_CAL_THRESHOLD) && (gAIL < - RC_CAL_THRESHOLD) && (gRUD
    > RC_CAL_THRESHOLD))
205   {
206     rc_enable_motor = 1;
207     fly_ready = 1;
208   }
209}
210
211/*
212 * Convert RC received gAIL, gELE, gRUD
213 */
214void GetTargetEulerAngle(EulerAngleTypeDef *euler_rc, EulerAngleTypeDef *euler_ahrs)
215{
216t1 = gELE;
217     if (t1 > RC_FULLSCALE)
218         t1 = RC_FULLSCALE;
219     else if (t1 < -RC_FULLSCALE)
220         t1 = - RC_FULLSCALE;
221     euler_rc->thx = -t1 * max_pitch_rad / RC_FULLSCALE;
222
223     t1 = gAIL;
224     if (t1 > RC_FULLSCALE)
225         t1 = RC_FULLSCALE;
226     else if (t1 < -RC_FULLSCALE)
227         t1 = - RC_FULLSCALE;
228     euler_rc->thy = -t1 * max_roll_rad / RC_FULLSCALE;
229
230     t1 = gRUD;
231     if (t1 > RC_FULLSCALE)
232         t1 = RC_FULLSCALE;
233     else if (t1 < -RC_FULLSCALE)
234         t1 = - RC_FULLSCALE;
235
236     if(rc_z_control_flag == 1)
237     {
238       if(t1 > EULER_Z_TH)
239       {
240         euler_rc->thz = euler_rc->thz + max_yaw_rad;
241       }
242       else if(t1 < -EULER_Z_TH)
243       {
244         euler_rc->thz = euler_rc->thz - max_yaw_rad;
245       }
246     }
247     else
```

特定のスティック操作（ELE と THR が下端、AIL が左端、RUD が右端）があったとき、センサデータのオフセットを再取得する処理を起動する。

特定のスティック操作（ELE と THR が下端、AIL が右端、RUD が左端）があったとき、モータを動作可能状態とする。

エレベータスティック操作量からピッチ姿勢角度目標値を作る。

エルロンスティック操作量からロール姿勢角度目標値を作る。

ラダースティック操作量から方向目標値を作る。
（～253 行目）

```
248     {
249       if(t1 > -EULER_Z_TH&&t1 < EULER_Z_TH)
250       {
251            rc_z_control_flag = 1;
252       }
253     }
254}
255
256
257void init_queue(Queue_TypeDef *q)
258{
259  int32_t i;
260
261  q->header = 0;
262  q->tail = 0;
263  q->length = QUEUE_LENGTH;
264  q->full = 0;
265  q->empty = 1;
266  for (i=0;i<QUEUE_LENGTH;i++)
267  {
268    q->buffer[i][0] = 0;
269    q->buffer[i][1] = 0;
270  }
271}
272
273void add_queue(Queue_TypeDef *q, int16_t idx, int16_t value)
274{
275  int h;
276  cnt++;
277  if (q->full==1)
278  {
279    if (q->header == 0)
280      h = q->length - 1;
281    else
282      h = q->header - 1;
283    q->buffer[h][0] = idx;
284    q->buffer[h][1] = -1;   // set error flag
285  }
286  else
287  {
288    // insert the node
289    q->buffer[q->header][0] = idx;
290    q->buffer[q->header][1] = value;
291    // increase the header pointer
292    q->header++;
293    if (q->header >= q->length)
294      q->header = 0;
295    // check if queue is full
296    if (q->header == q->tail)
297      q->full = 1;
```

```
298     // it will not empty any more
299     q->empty = 0;
300   }
301 }
302
303 int32_t get_queue(Queue_TypeDef *q, int16_t *idx, int16_t *value)
304 {
305   // get data only if the queue is not empty
306   if (q->empty != 1)
307   {
308     // get the node data
309     *idx = q->buffer[q->tail][0];
310     *value = q->buffer[q->tail][1];
311     // moving tail pointer
312     q->tail++;
313     if (q->tail >= q->length)
314       q->tail = 0;
315     // check if queue is empty
316     if (q->header == q->tail)
317       q->empty = 1;
318     // It will not full any more
319     q->full = 0;
320     return 0;
321   }
322   else
323     return -1;      // queue is empty
324 }
325
326
327
```