```c
/*******************************************************************************
 *
 *    ________  _________  ____    ____  _____          ________   ______
 *   |_   __  ||  _   _  ||_   \   /  _||_   _|        |_   __  |.' ____ \
 *     | |_ \_||_/ | | \_|  |   \/   |    | |     ______ | |_ \_|| (____\_|
 *     |  _| _     | |      | |\  /| |    | |   _|______||  _| _  _.____`.
 *    _| |__/ |   _| |_    _| |_\/_| |_  _| |__/ |      _| |__/ || \____) |
 *   |________|  |_____|  |_____||_____||________|     |________| \______.'
 *
 *******************************************************************************
 *
 * File          : TLC5973.c
 * Version       : 1.0
 *
 *******************************************************************************
 *
 * Description   : Interface and data conversion for TLC5973
 *                 Number of drivers in serie has to be defined by user
 *                 A cycle represent a bit encoded as defined by datasheet
 *
 *******************************************************************************
 *
 * Author        : Miguel Santos
 * Date          : 14.09.2023
 *
 *******************************************************************************
 *
 * MPLAB X       : 5.45
 * XC32          : 2.50
 * Harmony       : 2.06
 *
 ******************************************************************************/

#include "TLC5973.h"
#include "SerialTimer.h"

/******************************************************************************/

/* Number of drivers connected in series */
#define DRIVER_COUNT 3

/*----------------------------------------------------------------------------*/

/* Number of channels per driver */
#define CHANNEL_COUNT 3

/* Single field cycles count */
#define FLD_CYCLE_COUNT 12

/* Sequences cycles count */
#define DWS_CYCLE_COUNT 48
#define DWS_TOTAL_COUNT (DWS_CYCLE_COUNT * DRIVER_COUNT)
#define EOS_CYCLE_COUNT 4
#define EOS_TOTAL_COUNT (EOS_CYCLE_COUNT * (DRIVER_COUNT - 1))
#define GSL_CYCLE_COUNT 10

/* Size of the buffer to store cycles */
#define TLC_BUFFER_SIZE (DWS_TOTAL_COUNT + EOS_TOTAL_COUNT + GSL_CYCLE_COUNT)

/* Buffer offset for each driver */
#define DWS_OFFSET (DWS_CYCLE_COUNT + EOS_CYCLE_COUNT)

/* Each driver starts with a write command */
#define WRITE_COMMAND 0x3AA

/* Cycles values encoding */
#define CYCLE_CODE_HIGH 0x05
#define CYCLE_CODE_LOW  0x01
#define CYCLE_CODE_SKIP 0x00

/* Mask to MSB bit in field */
#define FIELD_MASK 0x800

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

```
74
75  /* Struct of a single channel */
76  typedef struct
77  {
78      /* Output value of the channel */
79      uint16_t out;
80
81      /* Pointers where values will be stored in CYCLE buffer */
82      uint8_t *p_out;
83
84      /* Flag new value set on channel */
85      bool newValue;
86
87  }S_TLC_CHANNEL;
88
89  /* Struct of a single driver */
90  typedef struct
91  {
92      /* Each driver has 3 output channel */
93      S_TLC_CHANNEL channel[CHANNEL_COUNT];
94
95      /* Flag new value set on driver */
96      bool newValue;
97
98  }S_TLC_DRIVER;
99
100 /*****************************************************************************/
101
102 S_TLC_CHANNEL tlcCommands[DRIVER_COUNT];
103
104 S_TLC_DRIVER tlcDrivers[DRIVER_COUNT];
105
106 uint8_t cyclesBuffer[TLC_BUFFER_SIZE];
107
108 /*****************************************************************************/
109
110 static bool TLC_SetChannel(S_TLC_CHANNEL *channel, uint16_t value);
111 static bool TLC_TranslateChannel(S_TLC_CHANNEL *channel);
112 static bool TLC_TranslateDriver(S_TLC_DRIVER *driver);
113 static bool TLC_TranslateAll( void );
114
115 /*****************************************************************************/
116
117 void TLC_Initialize( void )
118 {
119     uint8_t i_buff;
120     uint8_t i_drv;
121     uint8_t i_cha;
122
123     /* Initialize TLC_Buffer with CYCLE_SKIP */
124     for(i_buff = 0; i_buff < TLC_BUFFER_SIZE; i_buff++ )
125     {
126         cyclesBuffer[i_buff] = CYCLE_CODE_SKIP;
127     }
128
129     /* Initialize TLC_Drivers with default values */
130     for(i_drv = 0; i_drv < DRIVER_COUNT; i_drv++ )
131     {
132         /* Commands are static channels in buffer */
133         tlcCommands[i_drv].out = WRITE_COMMAND;
134         tlcCommands[i_drv].p_out = cyclesBuffer +
135                                    (DWS_OFFSET * i_drv);
136         tlcCommands[i_drv].newValue = true;
137         TLC_TranslateChannel(&tlcCommands[i_drv]);
138
139         for(i_cha = 0; i_cha < CHANNEL_COUNT; i_cha++)
140         {
141             tlcDrivers[i_drv].channel[i_cha].out = 0x00;
142             tlcDrivers[i_drv].channel[i_cha].newValue = true;
143             tlcDrivers[i_drv].channel[i_cha].p_out = cyclesBuffer +
144                                                      (DWS_OFFSET * i_drv) +
145                                                      (FLD_CYCLE_COUNT * (i_cha + 1));
146         }
```

```
147             tlcDrivers[i_drv].newValue = true;
148         }
149 
150         TLC_TranslateAll();
151 
152         /* Initialize the serial timer */
153         STR_Init();
154     }
155     /*****************************************************************************/
156 
157     bool TLC_Transmit( void )
158     {
159         bool status;
160 
161         status = TLC_TranslateAll();
162 
163         if(status)
164         {
165             STR_AddBuffer(cyclesBuffer, TLC_BUFFER_SIZE);
166             STR_Start();
167         }
168 
169         return status;
170     }
171 
172     /*****************************************************************************/
173 
174     bool TLC_SetAll(uint16_t out0, uint16_t out1, uint16_t out2)
175     {
176         bool status;
177         uint8_t i_drv;
178 
179         status = false;
180 
181         for(i_drv = 0; i_drv < DRIVER_COUNT; i_drv++)
182         {
183             status &= TLC_SetDriver(i_drv, out0, out1, out2);
184         }
185 
186         return status;
187     }
188 
189     /*****************************************************************************/
190 
191     bool TLC_SetDriver(E_TLC_DRV_ID driver, uint16_t out0, uint16_t out1, uint16_t out2)
192     {
193         bool status;
194 
195         status = false;
196 
197         if(driver < DRIVER_COUNT)
198         {
199             /* Set channel 0 */
200             status |= TLC_SetChannel(&tlcDrivers[driver].channel[0], out0);
201             /* Set channel 1 */
202             status |= TLC_SetChannel(&tlcDrivers[driver].channel[1], out1);
203             /* Set channel 2 */
204             status |= TLC_SetChannel(&tlcDrivers[driver].channel[2], out2);
205 
206             tlcDrivers[driver].newValue = status;
207         }
208 
209         return status;
210     }
211 
212     /*****************************************************************************/
213 
214     static bool TLC_SetChannel(S_TLC_CHANNEL *channel, uint16_t value)
215     {
216         bool status;
217 
218         /* Detect if there's a new value */
219         status = (channel->out != value);
```

```
220
221         if(status)
222         {
223             channel->out = value;
224             channel->newValue = true;
225         }
226
227         return status;
228     }
229
230     /*****************************************************************************/
231
232     static bool TLC_TranslateAll( void )
233     {
234         bool status;
235         uint8_t i_drv;
236
237         status = false;
238
239         for(i_drv = 0; i_drv < DRIVER_COUNT; i_drv++)
240         {
241             if(tlcDrivers[i_drv].newValue)
242             {
243                 status |= TLC_TranslateDriver(&tlcDrivers[i_drv]);
244                 tlcDrivers[i_drv].newValue = false;
245             }
246         }
247
248         return status;
249     }
250
251     /*****************************************************************************/
252
253     static bool TLC_TranslateDriver(S_TLC_DRIVER *driver)
254     {
255         bool status;
256         uint8_t i_cha;
257
258         status = true;
259
260         for(i_cha = 0; i_cha < CHANNEL_COUNT; i_cha++)
261         {
262             if(driver->channel[i_cha].newValue)
263             {
264                 status &= TLC_TranslateChannel(&driver->channel[i_cha]);
265                 driver->channel[i_cha].newValue = false;
266                 driver->newValue = true;
267             }
268         }
269
270         return status;
271     }
272
273     /*****************************************************************************/
274
275     static bool TLC_TranslateChannel(S_TLC_CHANNEL *channel)
276     {
277         bool status;
278         uint8_t i_bits;
279
280         status = false;
281
282         /* Watchdog */
283         if(channel->p_out != NULL)
284         {
285             for(i_bits = 0 ; i_bits < FLD_CYCLE_COUNT ; i_bits++)
286             {
287                 if( channel->out & (FIELD_MASK >> i_bits) )
288                 {
289                     channel->p_out[i_bits] = CYCLE_CODE_HIGH;
290                 }
291                 else
292                 {
```

```
                channel->p_out[i_bits] = CYCLE_CODE_LOW;
            }
        }
        status = true;
    }

    return status;
}

/* ***************************************************************************
 End of File
 */

```