```c
/*
 * adc_driver.c
 *
 *  Created on: Dec 10, 2018
 *      Author: Dominic Doty
 */


/* HEADER */
#include "adc_driver.h"


/* STATIC FUNCTION DECLARATIONS */
static bool adc_null_ptrs(adc_init_config* config);
static bool adc_incompatible_mode(adc_channel channel, adc_bits bits);


/* FUNCTION DEFINITIONS */

// Initialize the ADC with set parameters
adc_error adc_init(adc_init_config* config)
{
    // Init Variables
    adc_error ret = ADC_ERROR_SUCCESS;

    // Check validity of config struct
    if(adc_null_ptrs(config))
    {
        ret = ADC_ERROR_NULL_PTR;
    }
    // Check valid mode (diff modes with diff channels)
    else if(adc_incompatible_mode(config->channel, config->bits))
    {
        ret = ADC_ERROR_CHANNEL_MODE_INCOMPATIBLE;
    }
    else if(config->adc != ADC0)
    {
        ret = ADC_ERROR_UNKNOWN_ADC;
    }
    else
    {
        // Easy Read Address
        ADC_Type* adc = config->adc;

        // GPIO Setup
        clock_ip_name_t kclock = ((((uint32_t)(config->port)) >> 12) & 0xFU) + 0x10380000U;
        CLOCK_EnableClock(kclock);
        PORT_SetPinMux(config->port, config->pin_1, kPORT_PinDisabledOrAnalog);
        PORT_SetPinMux(config->port, config->pin_2, kPORT_PinDisabledOrAnalog);

        // Enable Clock To Peripheral
        CLOCK_EnableClock(kCLOCK_Adc0);

        // CFG1
        adc->CFG1 = ADC_CFG1_ADLPC(config->low_power)                                   |
                    ADC_CFG1_ADIV(config->clock_div)                                    |
                    ADC_CFG1_ADLSMP(ADC_SAMP_CYCLE_ADDER_ADLSMP(config->sample_cycle_add))    |
                    ADC_CFG1_MODE(ADC_BITS(config->bits))                               |
                    ADC_CFG1_ADICLK(config->clock);

        // CFG2
        adc->CFG2 = ADC_CFG2_MUXSEL(config->mux)                                        |
                    ADC_CFG2_ADACKEN(config->async_state)                               |
                    ADC_CFG2_ADHSC(ADC_SAMP_CYCLE_ADDER_ADHSC(config->sample_cycle_add))    |
                    ADC_CFG2_ADLSTS(ADC_SAMP_CYCLE_ADDER_ADLSTS(config->sample_cycle_add));

        // SC2 - Note, ADTRG is set after calibration at the end
        adc->SC2 =     ADC_SC2_COMP(config->compare_mode)    |
                    ADC_SC2_DMAEN(config->dma_mode)        |
                    ADC_SC2_REFSEL(config->ref_volt)    ;

```

```
72          // SC3
73          adc->SC3 =    ADC_SC3_ADCO(config->continuous)    |
74                      ADC_SC3_AVG(config->avg_samps)    ;
75
76          switch(config->compare_mode)
77          {
78              case ADC_COMPARE_DISABLED:
79                  break;
80
81              case ADC_COMPARE_LESS:
82                  adc->CV1 = ADC_CV1_CV(MAX(config->compare_1, config->compare_2));
83                  break;
84
85              case ADC_COMPARE_RANGE_EXCLUSIVE_INSIDE:
86                  adc->CV1 = ADC_CV1_CV(MAX(config->compare_1, config->compare_2));
87                  adc->CV2 = ADC_CV2_CV(MIN(config->compare_1, config->compare_2));
88                  break;
89
90              case ADC_COMPARE_GREATER:
91                  adc->CV1 = ADC_CV1_CV(MAX(config->compare_1, config->compare_2));
92                  break;
93
94              case ADC_COMPARE_RANGE_INCLUSIVE_INSIDE:
95                  adc->CV1 = ADC_CV1_CV(MIN(config->compare_1, config->compare_2));
96                  adc->CV2 = ADC_CV2_CV(MAX(config->compare_1, config->compare_2));
97                  break;
98
99              case ADC_COMPARE_RANGE_EXCLUSIVE_OUTSIDE:
100                 adc->CV1 = ADC_CV1_CV(MIN(config->compare_1, config->compare_2));
101                 adc->CV2 = ADC_CV2_CV(MAX(config->compare_1, config->compare_2));
102                 break;
103
104             case ADC_COMPARE_RANGE_INCLUSIVE_OUTSIDE:
105                 adc->CV1 = ADC_CV1_CV(MAX(config->compare_1, config->compare_2));
106                 adc->CV2 = ADC_CV2_CV(MIN(config->compare_1, config->compare_2));
107                 break;
108         }
109
110         // Calibrate
111         adc->SC3 |= ADC_SC3_CAL(1);
112         while(!(*(adc->SC1) & ADC_SC1_COCO_MASK));    // Wait for cal to complete
113
114         // Check for Failure
115         if(adc->SC3 & ADC_SC3_CALF_MASK)
116         {
117             ret = ADC_ERROR_FAILED_CAL;
118         }
119         else
120         {
121             // Set up Interrupt Flag, Trigger
122             if(config->interrupt == ADC_INT_ON_COMPLETE)
123             {
124                 adc->SC2 |= ADC_SC2_ADTRG(config->trigger);
125                 NVIC_EnableIRQ(ADC0_IRQn);
126             }
127
128             // SC1 set channel (also starts conversion)
129             adc->SC1[0] =     ADC_SC1_ADCH_DIFF(config->channel) |
130                             ADC_SC1_AIEN(config->interrupt);
131         }
132     }
133
134     return ret;
135 }
136
137 // Start an ADC Conversion - Only needed in single shot mode, init automatically starts continuous mode
138 void adc_start_conversion(ADC_Type* adc, adc_mux_select mux, adc_channel channel)
139 {
140     adc->SC1[mux] = ((adc->SC1[mux]) & ~(ADC_SC1_ADCH_MASK | ADC_SC1_DIFF_MASK)) | channel;
141 }
142
143 // Get result blocking
```

```
144 uint16_t adc_blocking_result(ADC_Type* adc, adc_mux_select mux, adc_bits bits)
145 {
146     uint16_t mask_lut[] = ADC_BITS_RESULT_MASK_LUT;
147     while(!(adc->SC1[mux] & ADC_SC1_COCO_MASK));
148     return adc->R[mux] & mask_lut[bits];
149 }
150
151 // Calculate the sample rate based on supplied clock parameters
152 // Not meant for use in normal application, used for debugging
153 uint32_t adc_sample_rate_calc(adc_init_config* config)
154 {
155     // Initialize
156     uint32_t sample_rate = 0;
157
158     // Find Clock rate based on source, (bus, bus/2, alt, async), calc the div freq
159     uint32_t clock_rate = 0;
160     switch(config->clock)
161     {
162         case ADC_CLOCK_SEL_BUS:
163             clock_rate = CLOCK_GetBusClkFreq();
164             break;
165         case ADC_CLOCK_SEL_BUSDIV2:
166             clock_rate = CLOCK_GetBusClkFreq()/2;
167             break;
168         case ADC_CLOCK_SEL_ALTCLK:
169             clock_rate = 0; // Cannot find info on Alt freq
170             break;
171         case ADC_CLOCK_SEL_ADACK:;
172             // Typical Frequency Values - page 29 of datasheet
173             uint32_t ADACK_freqs[] = ADACK_FREQUENCY_LUT;
174             uint8_t ADACK_index = ADACK_FREQUENCY_LUT_INDEX((config->low_power), (config->sample_cycle_add >> 3));
175             clock_rate = ADACK_freqs[ADACK_index];
176             break;
177         default:
178             break;
179     }
180
181     // Calculate Divided Clock Rate
182     clock_rate /= (1 << config->clock_div);
183
184
185     // Calculate Clocks/Sample
186     uint8_t average_number[] = ADC_SAMP_AVERAGE_LUT;
187     uint8_t mode_base_cycles[] = ADC_BITS_BASE_CYCLE_LUT;
188     uint8_t long_mode_cycles[] = ADC_SAMP_CYCLE_ADDER_LUT;
189
190     // Find Clock/Samp based on Average Mode, Convert Mode, Long Sample Time Adder, and High Speed Time Adder
191     // Note that this ignores Single/First Continuous Time Add
192     uint16_t clocks_per_sample =(average_number[config->avg_samps])            *
193                                 ((mode_base_cycles[config->bits]) + (long_mode_cycles[config->sample_cycle_add]));
194
195
196     // Calculate Sample Rate
197     sample_rate = clock_rate/clocks_per_sample;
198
199
200     // Check that we're within clock limits
201     if(    (config->bits == ADC_BITS_16BIT) |
202         (config->bits == ADC_BITS_16BIT_DIFF))
203     {
204         if(clock_rate > 12000000)
205         {
206             sample_rate = 0;     // 16 bit can go 12MHz clock source
207         }
208     }
209     else
210     {
211         if(clock_rate > 18000000)
212         {
213             sample_rate = 0;    // 13 bit or less can go 18MHz clock source
214         }
215     }
```

```
216
217
218     return sample_rate;
219 }
220
221
222 /* STATIC FUNCTION DEFINITIONS */
223
224 // Check for null pointers in the config struct
225 static bool adc_null_ptrs(adc_init_config* config)
226 {
227     // Initialize
228     bool ret = false;
229
230     // Null config struct
231     if(    (config == NULL)         |    // Null Config struct
232         (config->adc == NULL)     |    // Null ADC pointer
233         (config->port == NULL)    )    // Null GPIO Port
234     {
235         ret = true;
236     }
237     return ret;
238 }
239
240 // Check that mode and selected channel are compatible (diff channel with diff mode)
241 static bool adc_incompatible_mode(adc_channel channel, adc_bits bits)
242 {
243     // Initialize
244     bool ret = false;
245
246     if(channel != ADC_CHAN_DISABLED)    // If channel is disabled, don't bother checking mode
247     {
248         bool diff_mode = (bits >= 0x4U);    // Diff Modes, Diff Channels
249         bool diff_channel =((channel == ADC_CHAN_DAD0)            |
250                             (channel == ADC_CHAN_DAD1)            |
251                             (channel == ADC_CHAN_DAD2)            |
252                             (channel == ADC_CHAN_DAD3)            |
253                             (channel == ADC_CHAN_TEMP_DIFF)        |
254                             (channel == ADC_CHAN_BANDGAP_DIFF)    |
255                             (channel == ADC_CHAN_VREFSH_DIFF));
256
257         ret = !(diff_mode == diff_channel);
258     }
259
260     return ret;
261 }
```