```
/*
  This is Time.cpp, the .cpp file for the Time library

  ------------------------------------------------------
  This code is found online. It was not written by team 26
  ------------------------------------------------------

  time.c - low level time and date functions
  Copyright (c) Michael Margolis 2009-2014


  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA

  1.0  6  Jan 2010 - initial release
  1.1  12 Feb 2010 - fixed leap year calculation error
  1.2  1  Nov 2010 - fixed setTime bug (thanks to Korman for this)
  1.3  24 Mar 2012 - many edits by Paul Stoffregen: fixed timeStatus() to update
                     status, updated examples for Arduino 1.0, fixed ARM
                     compatibility issues, added TimeArduinoDue and TimeTeensy3
                     examples, add error checking and messages to RTC examples,
                     add examples to DS1307RTC library.
  1.4  5  Sep 2014 - compatibility with Arduino 1.5.7
*/

#if ARDUINO >= 100
#include <Arduino.h>
#else
#include <WProgram.h>
#endif

#include "Time.h"

static tmElements_t tm;          // a cache of time elements
static time_t cacheTime;   // the time the cache was updated
static uint32_t syncInterval = 300;  // time sync will be attempted after this many
seconds

void refreshCache(time_t t) {
  if (t != cacheTime) {
    breakTime(t, tm);
    cacheTime = t;
  }
}

int hour() { // the hour now
  return hour(now());
}

int hour(time_t t) { // the hour for the given time
  refreshCache(t);
  return tm.Hour;
}

int hourFormat12() { // the hour now in 12 hour format
  return hourFormat12(now());
}

```

```
69  int hourFormat12(time_t t) { // the hour for the given time in 12 hour format
70    refreshCache(t);
71    if( tm.Hour == 0 )
72      return 12; // 12 midnight
73    else if( tm.Hour  > 12)
74      return tm.Hour - 12 ;
75    else
76      return tm.Hour ;
77  }
78
79  uint8_t isAM() { // returns true if time now is AM
80    return !isPM(now());
81  }
82
83  uint8_t isAM(time_t t) { // returns true if given time is AM
84    return !isPM(t);
85  }
86
87  uint8_t isPM() { // returns true if PM
88    return isPM(now());
89  }
90
91  uint8_t isPM(time_t t) { // returns true if PM
92    return (hour(t) >= 12);
93  }
94
95  int minute() {
96    return minute(now());
97  }
98
99  int minute(time_t t) { // the minute for the given time
100   refreshCache(t);
101   return tm.Minute;
102 }
103
104 int second() {
105   return second(now());
106 }
107
108 int second(time_t t) {  // the second for the given time
109   refreshCache(t);
110   return tm.Second;
111 }
112
113 int day(){
114   return(day(now()));
115 }
116
117 int day(time_t t) { // the day for the given time (0-6)
118   refreshCache(t);
119   return tm.Day;
120 }
121
122 int weekday() {   // Sunday is day 1
123   return  weekday(now());
124 }
125
126 int weekday(time_t t) {
127   refreshCache(t);
128   return tm.Wday;
129 }
130
131 int month(){
132   return month(now());
133 }
134
135 int month(time_t t) {  // the month for the given time
136   refreshCache(t);
137   return tm.Month;
```

```
}

int year() {  // as in Processing, the full four digit year: (2009, 2010 etc)
  return year(now());
}

int year(time_t t) { // the year for the given time
  refreshCache(t);
  return tmYearToCalendar(tm.Year);
}

/*=====================================================================*/
/* functions to convert to and from system time */
/* These are for interfacing with time serivces and are not normally needed in a sketch
*/

// leap year calulator expects year argument as years offset from 1970
#define LEAP_YEAR(Y)     ( ((1970+(Y))>0) && !((1970+(Y))%4) && ( ((1970+(Y))%100) || !((1970+(Y))%400) ) )

static  const uint8_t monthDays[]={31,28,31,30,31,30,31,31,30,31,30,31}; // API starts months from 1, this array starts from 0

void breakTime(time_t timeInput, tmElements_t &tm){
// break the given time_t into time components
// this is a more compact version of the C library localtime function
// note that year is offset from 1970 !!!

  uint8_t year;
  uint8_t month, monthLength;
  uint32_t time;
  unsigned long days;

  time = (uint32_t)timeInput;
  tm.Second = time % 60;
  time /= 60; // now it is minutes
  tm.Minute = time % 60;
  time /= 60; // now it is hours
  tm.Hour = time % 24;
  time /= 24; // now it is days
  tm.Wday = ((time + 4) % 7) + 1;  // Sunday is day 1

  year = 0;
  days = 0;
  while((unsigned)(days += (LEAP_YEAR(year) ? 366 : 365)) <= time) {
    year++;
  }
  tm.Year = year; // year is offset from 1970

  days -= LEAP_YEAR(year) ? 366 : 365;
  time  -= days; // now it is days in this year, starting at 0

  days=0;
  month=0;
  monthLength=0;
  for (month=0; month<12; month++) {
    if (month==1) { // february
      if (LEAP_YEAR(year)) {
        monthLength=29;
      } else {
        monthLength=28;
      }
    } else {
      monthLength = monthDays[month];
    }

    if (time >= monthLength) {
      time -= monthLength;
    } else {
```

```
204           break;
205         }
206       }
207       tm.Month = month + 1;  // jan is month 1
208       tm.Day = time + 1;     // day of month
209     }
210
211     time_t makeTime(const tmElements_t &tm){
212     // assemble time elements into time_t
213     // note year argument is offset from 1970 (see macros in time.h to convert to other
      formats)
214     // previous version used full four digit year (or digits since 2000),i.e. 2009 was 2009
      or 9
215
216       int i;
217       uint32_t seconds;
218
219       // seconds from 1970 till 1 jan 00:00:00 of the given year
220       seconds= tm.Year*(SECS_PER_DAY * 365);
221       for (i = 0; i < tm.Year; i++) {
222         if (LEAP_YEAR(i)) {
223           seconds +=  SECS_PER_DAY;   // add extra days for leap years
224         }
225       }
226
227       // add days for this year, months start from 1
228       for (i = 1; i < tm.Month; i++) {
229         if ( (i == 2) && LEAP_YEAR(tm.Year)) {
230           seconds += SECS_PER_DAY * 29;
231         } else {
232           seconds += SECS_PER_DAY * monthDays[i-1];  //monthDay array starts from 0
233         }
234       }
235       seconds+= (tm.Day-1) * SECS_PER_DAY;
236       seconds+= tm.Hour * SECS_PER_HOUR;
237       seconds+= tm.Minute * SECS_PER_MIN;
238       seconds+= tm.Second;
239       return (time_t)seconds;
240     }
241     /*=====================================================*/
242     /* Low level system time functions  */
243
244     static uint32_t sysTime = 0;
245     static uint32_t prevMillis = 0;
246     static uint32_t nextSyncTime = 0;
247     static timeStatus_t Status = timeNotSet;
248
249     getExternalTime getTimePtr;  // pointer to external sync function
250     //setExternalTime setTimePtr; // not used in this version
251
252     #ifdef TIME_DRIFT_INFO   // define this to get drift data
253     time_t sysUnsyncedTime = 0; // the time sysTime unadjusted by sync
254     #endif
255
256
257     time_t now() {
258       // calculate number of seconds passed since last call to now()
259       while (millis() - prevMillis >= 1000) {
260         // millis() and prevMillis are both unsigned ints thus the subtraction will always
          be the absolute value of the difference
261         sysTime++;
262         prevMillis += 1000;
263     #ifdef TIME_DRIFT_INFO
264         sysUnsyncedTime++; // this can be compared to the synced time to measure long term
          drift
265     #endif
266       }
267       if (nextSyncTime <= sysTime) {
268         if (getTimePtr != 0) {
```

```
269         time_t t = getTimePtr();
270         if (t != 0) {
271           setTime(t);
272         } else {
273           nextSyncTime = sysTime + syncInterval;
274           Status = (Status == timeNotSet) ?  timeNotSet : timeNeedsSync;
275         }
276       }
277     }
278     return (time_t)sysTime;
279   }
280
281   void setTime(time_t t) {
282   #ifdef TIME_DRIFT_INFO
283    if(sysUnsyncedTime == 0)
284      sysUnsyncedTime = t;   // store the time of the first call to set a valid Time
285   #endif
286
287     sysTime = (uint32_t)t;
288     nextSyncTime = (uint32_t)t + syncInterval;
289     Status = timeSet;
290     prevMillis = millis();  // restart counting from now (thanks to Korman for this fix)
291   }
292
293   void setTime(int hr,int min,int sec,int dy, int mnth, int yr){
294    // year can be given as full four digit year or two digts (2010 or 10 for 2010);
295    //it is converted to years since 1970
296     if( yr > 99)
297         yr = yr - 1970;
298     else
299         yr += 30;
300     tm.Year = yr;
301     tm.Month = mnth;
302     tm.Day = dy;
303     tm.Hour = hr;
304     tm.Minute = min;
305     tm.Second = sec;
306     setTime(makeTime(tm));
307   }
308
309   void adjustTime(long adjustment) {
310     sysTime += adjustment;
311   }
312
313   // indicates if time has been set and recently synchronized
314   timeStatus_t timeStatus() {
315     now(); // required to actually update the status
316     return Status;
317   }
318
319   void setSyncProvider( getExternalTime getTimeFunction){
320     getTimePtr = getTimeFunction;
321     nextSyncTime = sysTime;
322     now(); // this will sync the clock
323   }
324
325   void setSyncInterval(time_t interval){ // set the number of seconds between re-sync
326     syncInterval = (uint32_t)interval;
327     nextSyncTime = sysTime + syncInterval;
328   }
```