Swfltek Time Library

Introduction

This work is designed as a ‘supplement’ to avr-libc, providing the functionality described in ISO/IEC 9899, section 7.23 (C90 <time.h>)

The C standards make a distinction between 'hosted' and 'free standing' C library implementations. While a free standing C implementation need only conform to <float.h>, <iso646.h>, <limits.h>, <stdarg.h>, <stdbool.h>, <stddef.h>, and <stdint.h>, several other standard C headers are usually implemented as well.

In avr-libc, for example, additional headers are...

> <assert.h>, <ctype.h>, <errno.h>, <inttypes.h>, <math.h>,
> <setjmp.h>, <stdio.h>, <stdlib.h>, <string.h>

These ‘supplements’ to the required set vary in how strongly they conform to the standard. <ctype.h> usually adheres strongly, since there is really no architecture or platform issues.

The avr-libc <stdio.h> does not adhere as well to the standard, due to architecture and platform. Yet it still provides useful functionality, which is ‘close enough’ to the standard that the typical programmer can get started pretty quickly, and without having to learn (yet another) non-standard API.

This work attempts to adhere to the standard where possible and practical.

As with the avr-libc <stdio.h> implementation, the characteristics of the ATMEL™ AVR 8 bit mcu and of the avr-gcc tool chain necessitate deviations from the standard. Where such deviations are required, every attempt is made to provide functional equivalency.

Base Functions and Stuctures

Refer to the standards document (ISO/IEC 9899) for details of how these functions are defined to operate. In this section, we will detail only the departures from the standard, and any other useful information.

typdef uint32_t time_t;

Seconds since midnight GMT, Jan 1, 2000. This is supposedly an ‘opaque’ type.

time_t time(time_t * timer);

See the Usage chapter.

/*clock_t, CLOCKS_PER_SECOND, and clock();*/

This functionality properly belongs to an operating system. Therefore declared as ‘out of scope’ of this library.

```
struct tm{
        int8_t tm_sec;
        int8_t tm_min;
        int8_t tm_hour;
        int8_t tm_mday;
        int8_t tm_wday;
        int8_t tm_mon;
        int16_t tm_year;
        int16_t tm_yday;
        int16_t tm_isdst;
};
```

The standard specifies each field is an ‘int’, without specifying sizeof(int). We save a few bytes of RAM, by using 8 bit values where possible.

Another slight variation from the standard is the usage of the field ‘tm_isdst’. Beyond the standard, we specify that when DST is in effect, tm_isdst will equal the number of seconds of clock advancement.

int32_t difftime(time_t time1, time_t time0);

The standard difftime() returns a double. Since we do not have a true double to work with, we return a ‘work alike’ value.

time_t mktime(struct tm *timeptr);

struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
Use of these functions will result in the permanent allocation of a 12 byte private struct tm object.

Use the ‘re entrant’ versions to avoid that allocation.

char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);

The use of these functions will cause the permanent allocation of a 26 byte private buffer, shared between them and isotime()

Use the ‘re entrant’ versions to avoid that allocation.

In gcc versions prior to 4.7, these functions will also allocate a further 83 bytes for strings. In 4.7 and later, the strings will be assigned to FLASH.

size_t strftime(char * s, size_t maxsize, const char * format, const struct tm * timeptr);

All conversions are performed as if operating in the ‘C locale’. The E and O modifiers are silently ignored.

This function is easily the largest, most complex, and most RAM hungry function in the library. In gcc versions before 4.7, some 250 bytes of RAM will be consumed for strings. In version 4.7 and later this is reduced to around 100 bytes.

Re-Entrant version of standard functions
Not specified in the standard, but very handy to have. Especially when every byte of RAM counts.

void asctime_r(const struct tm *timeptr, char * buffer);

void ctime_r(const time_t *timer, char * buffer);

void gmtime_r(const time_t * timer, struct tm *timeptr);

void localtime_r(const time_t * timer, struct tm *timeptr);

Support functions

These additional functions are not required by the standard, but are needed to make the library actually work.

void set_zone(long utc_offset);

Defines the Time Zone, in seconds East of the prime meridian.

Western zones are therefore negative.

America/New York, for example, is -18000 seconds (-5 * ONE_HOUR)

void set_dst(int16_t (*dst_function)(time*t timer, long utc_offset));

Set the 'callback function' used to compute Daylight Savings. If this is not done, the library will ignore Daylight Savings.

The dst function is expected to return a value suitable for tm_isdst, that is, zero if DST is not in effect, the number of seconds to advance the clock if it is in effect, or -1 if it cannot be determined.

Included in the library are header files which implement DST rules for the United States and the European Union (effective as of the date of this writing).

To use them, just

```
#include <util/usa_dst.h>
        or
#include <util/eu_dst.h>
```

In the event you wish to write your own dst function, these files provide good examples.

void set_system_time(time_t systime);

Use this function to set the system clock.
See the Usage chapter for more details

void system_tick(void);

This function must be called at 1 second intervals.
See the Usage chapter for more details

Handy Extras

char * isotime(struct tm * timeptr);
void isotime_r(struct tm * timeptr, char * buffer);

Constructs a string representation of time in ISO8601 format.

Like asctime(), isotime() will use the shared 26 byte buffer... use the re entrant version to avoid that allocation.

Unlike asctime, no further RAM, or FLASH, is allocated for strings.

time_t mk_gmtime(struct tm * timeptr);

This is the equivalent of mktime(), where timeptr is in UTC rather than local time.

uint8_t is_leap_year(int16_t year);

Returns non-zero if the passed year is a leap year.

uint8_t length_of_month(int16_t year, uint8_t month);

Returns the length of the month, 28 to 31 days.
The year parameter is unrestricted, while month should be 1 through 12.

uint8_t week_of_month(const struct tm * timeptr, uint8_t start);
uint8_t week_of_year(const struct tm * timeptr, uint8_t start);

Returns the week number, based on the week starting on a given day of week (0 through 6).

Returns zero if that day of week has not yet occurred in the time frame.

```
struct week_date{
    int year;
    int week;
    int day;
};
```

Structure representing a week based date.

struct week_date iso_week_date(int year, int yday);
void iso_week_date_r(int year, int yday, struct week_date *);

Compute the ISO 8601 week based date. See See http://en.wikipedia.org/wiki/ISO_week_date for a full description.

uint32_t fat_time(tm * timeptr);

Returns a timestamp formatted for use with the FatFS file system driver.

#define ONE_HOUR 3600
#define ONE_DAY 86400

Number of seconds in one hour and one day.

#define UNIX_OFFSET 946684800

The number of seconds between the UNIX epoch and the Y2K epoch. To convert a UNIX time stamp to a Y2K time stamp, subtract this value.

#define NTP_OFFSET 3155673600

The number of seconds between the NTP epoch and the Y2K epoch.
To convert a NTP time stamp to a Y2K time stamp, subtract this value.

Ephemera

While not even close to being in any standard library, these functions are also very handy to have around.

void set_position(int32_t latitude, int32_t longitude);

Many of the 'ephemeral' functions require the geographic coordinates of the observer to be known.

Parameters are given in seconds of arc, North and East.
Negative values are therefore given for West or South.

int16_t equation_of_time(time_t * timer);

This is basically the difference between the time shown on a sundial, and that shown by a clock. The returned value is in seconds.

double solar_declination(time_t * timer);

Solar declination in radians.

int32_t daylight_seconds(time_t * timer);

Returns the length of time the sun is above the horizon.
This will range between 0 and 86400 seconds.
0 indicates the sun never rises that day.
86400 indicates the sun never sets that day.

time_t solar_noon(time_t * timer);

Returns the Local Time of solar noon at the observers location. The sun is at its highest altitude at this time.

time_t sun_rise(time_t * timer);
time_t sun_set(time_t * timer);

Returns the time of sunrise or sunset at the location of interest.
Inside the polar circles, this can be wildly wrong, it is recommended to check daylight_seconds.

int8_t moon_phase(time_t * timer);

Returns a rough approximation to the phase of the moon.
The returned magnitude represents the percentage of the surface which is sunlit.
The returned sign indicates if the moon is waxing (+) or waning (-).

unsigned long gm_sidereal(const time_t * timer);
unsigned long lm_sidereal(const time_t * timer);

Returns sidereal time in seconds.

Usage

The simplest, quickest way to get started...

Specify your time zone with set_zone()

Specify the Daylight Savings function with set_dst(), if you wish to account for Daylight Savings.

Arrange for system_tick() to be called once a second.

Set the system time. If using a Clock / Calendar type RTC, fill in the elements of a struct tm object, setting tm_isdst to zero. Use the result of mktime() as the system time...

```
time_t systime;
struct tm rtctime;

     read_rtc(&rtctime); // 'fills in' rtctime
     systime = mktime( &rtctime );
     set_system_time(systime);
```

It is anticipated that system_tick will be called from within an Interrupt Service Routine. Calling a function from within a normal ISR will cause a 'register' flush, where most cpu registers are saved on the stack before the function call, and restored afterward.

AVR-LIBC provides for 'naked' Interrupt Service Routines. When using a Naked ISR, it is up to the application to ensure that cpu registers are restored, and to return with the 'reti()' function. The application may thus avoid a register flush, by saving and restoring only the registers which are modified.

system_tick() is written to make this easy, and may be called from within a naked ISR using code similar to the following...

```
ISR(TIMER2_OVF_vect, ISR_NAKED){

     system_tick();
     reti;

}
```

FAQ

Isn't the Unix epoch standard?

No. The standard states that the encoding, range, and precision of time_t is implementation defined.

Though the UNIX epoch (Jan 1, 1970) is widely used, there are in fact a great variety of epochs in use. Probably the most widely used epoch is Jan 1, 1601, used by Microsoft Windows.

Why the Y2K epoch?

We chose Y2K for its mathematical significance. It is a conjunction of the 400, 100, and 4 year Leap day cycles. Using this property allows for faster math, and smaller code.

Doesn't the C standard specify time_t is a signed value?

No. The standard says time_t must be an 'arithmetic type' capable of representing time.

This means that, as long as mathematical operations on time_t will produce sensible results, the representation is valid under the standard.

It is perfectly valid (under the standard) for time_t to be char, int, long, or even a floating point type.

In fact, it could be construed to include complex types and vectors.

What is the range of this library?

As far as this library is concerned, time began January 1 2000

    2000-01-01 00:00:00

and will end on February 7 2136

    2136-02-07 01:28:15

So, a bit over 136 years.

What linker commands do I need to use?

None. The time functions install into the standard C library.

License and Copyright