AVR Threads Library 1.3

Dean Ferreyra
dean@octw.com
http://www.bourbonstreetsoftware.com/

November 11, 2008

Contents

Chapter 1

Introduction

The AVR Threads Library provides basic preemptive multi-threading to the Atmel AVR family of microcontrollers. It is written mostly in C with some assembly language. It implements a simple round-robin style task switcher.

This library provides basic thread start and stop functions, a couple of flavors of mutual exclusion semaphore, and events to help synchronize threads.

I have released this code under the GNU Lesser General Public License with the hope that others might find it useful.

Chapter 2

Build and Installation

To build and install the AVR Threads Library, you need to have already installed the AVR GCC cross-compiler.

2.1 Build

To install the AVR Threads Library, unpack the source tree. If you downloaded the `*.tgz` version, you can do this with the following command:

```
tar -xzf threads-1.3-src.tgz
```

Next, change directory to the head of the source tree that was just unpacked and run the `make` command:

```
$ cd threads-1.3
$ make
```

This should build the AVR Threads Library for all the supported AVR microcontrollers.

2.2 Installation

Once the libraries have been successfully built, you need to install the library components to a location where the AVR GCC compiler can find them.

The default installation directory prefix is `/usr/local`. If you wish to change this, you will have to edit the `src/Makefile` file in the source tree. In the Makefile, you'll find a line that looks like this:

```
prefix = /usr/local
```

Change this line to desired directory prefix. For example, if you're using WinAVR, you will need to change this this point to where WinAVR has been installed. For example, if WinAVR has been installed in `c:\WinAVR`, then change the prefix line to this:

```
prefix = c:/WinAVR
```

Finally, from the head of the source tree, or from the `src` directory, run this:

```
$ make install
```

Chapter 3

Basic Functions and Structures

3.1 Initializing the Library

3.1.1 Functions

`void avr_thread_init(void)` ***function***

This function initializes the AVR Threads Library. It must be called before any other AVR Threads Library function is called.

This function sets up a default context so that it can treat the currently running program as a thread. It also prepares the "idle" thread that runs when no other thread can run. After calling this function, task switches can occur, both implicit and explicit.

3.2 Starting, Stopping, and Pausing Threads

3.2.1 Functions

`void avr_thread_start(avr_thread_context* context, void (*fn)(void), uint8_t* stack, uint16_t stack_size)` ***function***

This function starts a new thread of execution given a pointer to a `avr_thread_context`, `context`; a pointer to a function where the thread will begin execution, `fn`; a pointer to a stack, `stack`; and the size of the stack, `stack_size`.

The memory pointed to by `context` must remain valid throughout the life of the new thread.

The stack space provided must also remain valid throughout the life of the new thread. The stack for each thread must be large enough to satisfy the

stack needs of the thread's code, plus enough additional space to hold the entire register state at the time of a task switch.

`void avr_thread_stop(void)` ***function***

This function stops the current thread. It does this by taking the current thread out of the task list and forcing a task switch.

`void avr_thread_sleep(uint16_t ticks)` ***function***

This function places the given thread to sleep for the given number of task-switcher ticks, `ticks`. The thread automatically awakens after the given time has elapsed.

Passing zero to this function forces a task switch and is equivalent to calling `avr_thread_yield()`.

`void avr_thread_yield(void)` ***inline***

This function forces an explicit task-switch away from the current thread. It is implemented by calling `avr_thread_sleep()` with an argument of zero.

3.3 Task Switcher

The library needs the help of a timer interrupt and its interrupt service routine to provide preemptive multitasking. The following two functions are designed to be called from within an interrupt service routine tagged with the `naked` compiler attribute. For example, here is the interrupt service routine for TIMER2 on the Atmel ATmega128 microcontroller:

```
void SIG_OUTPUT_COMPARE2(void) __attribute__((naked));
void SIG_OUTPUT_COMPARE2(void)
{
    /* Global interrupt can be re-enabled here if desired. */
    /* sei(); */
    avr_thread_isr_start();
    /* Place your normal ISR code here. */
    /* ... */
    /* This must be the last function called. */
    avr_thread_isr_end();
}
```

Also, here is an example configuration of TIMER2 that yields approximately 1kHz preemptive task switch frequency with a 16MHz crystal:

```
    /* ... */
    /* Setup TIMER2 mode.  Include reset on overflow bit. */
    /* Approximately 1 kHz for a 16 MHz crystal. */
```

```
TCCR2 = _BV(WGM21) | _BV(CS21) | _BV(CS20);
OCR2 = 250;
TCNT2 = 0;
TIMSK |= _BV(OCIE2);
/* Initialize library. */
avr_thread_init();
/* Enable global interrupts to start preemptive task switching */
sei();
/* ... */
```

3.3.1 Functions

`void avr_thread_isr_start(void)` ***function***

Call this function at the beginning of the interrupt service routine used to provide the preemptive multitasking. This function saves registers on the stack in preparation for a task switch. The interrupt service routine must be declared with the attribute "naked".

If you want to run your interrupt service routine code with global interrupts enabled, place a call to `sei()` just before this function. No other code should be placed ahead of the call to `avr_thread_isr_start()` since this will disturb the thread context before this function has a chance to save it.

`void avr_thread_isr_end(void)` ***function***

Call this function at the end of the interrupt service routine used to provide the preemptive multitasking. This function does not return.

Chapter 4

Thread Synchronization

The AVR Threads Library provides two basic ways to synchronize threads: mutual exclusion semaphores (mutexes) and events.

4.1 Mutexes

Mutual exclusion semaphores, or mutexes, provide a way to ensure that only one thread is executing a particular piece of code at a time.

The mutexes described here keep track of ownership and lock counts. This allows a thread to lock a mutex a number of times in the code path without having to worry about the thread blocking. Also, threads that are blocked on these mutexes are awakened in the order that they blocked so that the thread that has been waiting the longest will be awakened when the mutex become available.

4.1.1 Functions

`uint8_t avr_thread_mutex_gain(volatile avr_thread_mutex* mutex, uint16_t ticks)` ***function***

This function tries to gain ownership of the given mutex, `mutex`.

If the mutex is unlocked, the current thread locks the mutex and thereby gains ownership of the mutex, and this function returns `1` immediately.

If the mutex is already locked and owned by the current thread, the lock-count is incremented and the function returns `1` immediately.

If the mutex is already locked and owned by another thread, this thread blocks until the mutex becomes available or until the given number of ticks, `ticks`, have elapsed. If the thread successfully locks the mutex before the number of ticks have elapsed, the function return `1`. Otherwise, the function times-out and returns `0`. If the thread blocks and you have passed `ticks` as zero, the function will wait indefinitely until the mutex becomes available.

`void avr_thread_mutex_release(volatile avr_thread_mutex* mutex)` ***function***

This function releases the given mutex. The mutex is not really released until this function is called the same number of times as the `avr_thread_mutex_gain()` function was called in locking the mutex.

If multiple threads are waiting on this mutex, the thread that has been waiting the longest is released.

`struct avr_thread_mutex` ***structure***

This structure is used by the AVR Threads Library to hold mutex data. It should be treated as an opaque object.

4.2 Basic Mutexes

Basic mutexes are very simple and lack some important features of the regular mutexes (see 4.1). While the advantage of basic mutexes is their execution-speed and a smaller code and memory foot-print, you should exercise care in using them since they do not keep track of ownership or lock counts. For example, if a thread locks a basic mutex and tries to lock it again, the thread will block.

4.2.1 Functions

`void avr_thread_mutex_basic_gain(volatile avr_thread_mutex_basic* mutex)` ***function***

This function tries to lock the given basic mutex, `mutex`. If the mutex is already locked, the thread blocks on the basic-mutex waiting for another thread to release the lock on the thread.

A thread that has called `avr_thread_mutex_basic_gain()` should not call it again without first releasing the lock with `avr_thread_mutex_basic_release()`.

`uint8_t avr_thread_mutex_basic_test_and_gain(volatile avr_thread_mutex_basic* mutex)` ***function***

This function tries to lock the given basic mutex, `mutex`. If it is not already locked by another thread, the mutex is locked and the function returns `1`. If it is already locked by another thread, the function does not block but instead returns `0` immediately.

`void avr_thread_mutex_basic_release(volatile avr_thread_mutex_basic* mutex)` ***function***

This function releases the lock on the given basic mutex, `mutex`.

`struct avr_thread_mutex_basic` ***structure***

This structure is used by the AVR Threads Library to hold basic mutex data. It should be treated as an opaque object.

Chapter 5

Events

Events provide a way to signal to other threads that an event has occurred.

5.1 Functions

`void avr_thread_event_set_wake_one(volatile avr_thread_event* event)` ***function***

This function places the given event, `event` in a signaled state. If there are threads waiting on this event, the thread that has been waiting the longest is awakened.

While the event is signaled, a thread that calls `avr_thread_event_wait()` or `avr_thread_event_wait_and_clear()` will return immediately.

`void avr_thread_event_set_wake_all(volatile avr_thread_event* event)` ***function***

This function places the given event, `event` in a signaled state. If there are threads waiting on this event, all of them are awakened.

While the event is signaled, a thread that calls `avr_thread_event_wait()` or `avr_thread_event_wait_and_clear()` will return immediately.

`void avr_thread_event_clear(volatile avr_thread_event* event)` ***function***

This function clears the given event, `event`. While the event is clear, a thread that calls `avr_thread_event_wait()` or `avr_thread_event_wait_and_clear()` will block until the event is signaled.

`uint8_t avr_thread_event_wait(volatile avr_thread_event* event, uint16_t ticks)` ***function***

This function waits for the given event, `event`, to become signaled.

If the event is signaled when this function is called, this function returns `1` immediately. Otherwise, the thread is blocked until the event is signaled by another thread.

While blocked, if the thread is awakened by another thread through a call to `avr_thread_event_set_wake_all()` or `avr_thread_event_set_wake_one()`, then this function returns `1`. Otherwise, if the given number of ticks, `ticks`, elapses, the function times-out and returns `0`.

If you pass `ticks` as zero, this function will wait indefinitely until the event becomes signaled.

`uint8_t avr_thread_event_wait_and_clear(volatile avr_thread_event* event, uint16_t ticks)` ***function***

This function waits for the given event, `event`, to become signaled.

If the event is signaled when this function is called, this function clears the event and returns `1` immediately. Otherwise, the thread is blocked until the event is signaled by another thread.

While blocked, if the thread is awakened by another thread through a call to `avr_thread_event_set_wake_all()` or `avr_thread_event_set_wake_one()`, then this function clears the event and returns `1`. Otherwise, if the given number of ticks, `ticks`, elapses, the function times-out and returns `0`.

If you pass `ticks` as zero, this function will wait indefinitely until the event becomes signaled.

`struct avr_thread_event` ***structure***

This structure is used by the AVR Threads Library to hold event data. It should be treated as an opaque object.

Chapter 6

Additional Functions and Structures

6.1 Functions

`void avr_thread_enable(void)` ***inline***

This function re-enables task switching that has been disabled by `avr_thread_disable()`.

The `avr_thread_enable()` and `avr_thread_disable()` functions keep track of how many times they've been called. Task switching will only be enabled when `avr_thread_enable()` has been called as many times as `avr_thread_disable()` was called in disabling task switching.

`void avr_thread_disable(void)` ***inline***

This function disables task switching. Task switching is re-enabled by calling `avr_thread_enable()`.

The `avr_thread_enable()` and `avr_thread_disable()` functions keep track of how many times they've been called. Task switching will only be enabled when `avr_thread_enable()` has been called as many times as `avr_thread_disable()` was called in disabling task switching.

`struct avr_thread_context` ***structure***

This structure holds data used by the AVR Threads Library to manage each thread. It should be treated as an opaque object.

Chapter 7

Sample Program

Here is a sample program that demonstrates the basics of initializing the library and starting a thread. It was originally written for the ATmega128.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr-thread.h>

// Thread stack
uint8_t fn_stack[128];
// Thread context
avr_thread_context fn_context;
// Thread code
void fn(void)
{
    uint8_t state = 0;
    for (;;) {
        if (state)
            PORTB &= ~0x02;
        else
            PORTB |= 0x02;
        state = ! state;
    }
}

int main(void)
{
    // Setup port B as all output.
    PORTB = 0xff;
    DDRB = 0xff;

    // Setup timer 2 mode.  Include reset on overflow bit.
```

```
    // Approximately 1.008 kHz for 4 MHz crystal.
    TCCR2 = _BV(WGM21) | _BV(CS21) | _BV(CS20);
    OCR2 = 62;
    TCNT2 = 0;
    TIMSK |= _BV(OCIE2);

    // Initialize avr-thread library.
    avr_thread_init();
    sei();
    // Start new thread
    avr_thread_start(&fn_context,
                     fn, fn_stack, sizeof(fn_stack));

    uint8_t state = 0;
    for (;;) {
        if (state)
            PORTB &= ~0x01;
        else
            PORTB |= 0x01;
        state = ! state;
    }
}

uint32_t switch_count = 0;

// Task switcher
void SIG_OUTPUT_COMPARE2(void) __attribute__((naked));
void SIG_OUTPUT_COMPARE2(void)
{
    sei();
    avr_thread_isr_start();
    switch_count++;
    avr_thread_isr_end();
}
```

Index