Minimal Guide to ThreadKit and Muser

October 4, 2011

Introduction

`ThreadKit` [http://code.google.com/p/threadkit/] is a C++ library which provides simple, low-overhead, non-stack-based, cooperative multithreading on Arduino and similar microcontroller systems. This release of **`ThreadKit`** works in conjunction with the standard Arduino library (version 0022). It has also been compiled and tested using the Eclipse IDE with AVR plugin [http://www.arduino.cc/playground/Code/Eclipse] and directly using Makefiles.

`ThreadKit` was built from scratch and is self contained: it does not require the use of any supplemental libraries. The library was in part inspired by the work of Adam Dunkels on Protothreads [http://www.sics.se/~adam/pt/] and Larry Ruane on his more complex protothread library [http://code.google.com/p/protothread/]. The **`EVENT`** API is similar to the one used in the Scratch language [http://scratch.mit.edu/].

This document is preliminary and, as the title says, minimal; I hope to improve it in the near future with additional background information, tutorial content, real examples, and so on, and I welcome feedback.

Elements not included in this release because the macro and include-file techniques by which they are implemented are not compatible with the Arduino build process are:

- the "console" system, for setting and inspecting variables and invoking functions via a standard serial terminal
- the "monit" system, which enables the display of multiple sensor values in real-time on simulated oscilloscopes

Also not included in this release is:

- **`BehaviorKit`**, an extension for the implementation of behavior-based robots

To install **`ThreadKit`**, follow the standard Arduino procedure: simply copy the entire **`ThreadKit`** directory into the **`libraries`** subdirectory of your Sketchbook. In addition to reviewing this documentation, the best way to learn how to use **`ThreadKit`** is to study the example program, **`muser`**.

Basic Framework

`ThreadKit.cpp` includes its own **`setup()`** and **`loop()`** functions, so your main sketch file simply needs to have the line

```
#include "Threadkit.h"
```

By default, **`ThreadKit`** sets up serial communications with an initial serial speed determined by the constant **`SERIAL_RATE`**. This value is defined in **`ThreadKit.h`**, and is currently set to 57600.

If you need initialization code to run before any threads start, you can put it in the function **`user_setup()`** (instead of **`setup`**). The rest of your code goes in one or more thread functions.

Example Program

`muser` is a simple music program that demonstrates the use of **`ThreadKit`** threads and events by playing two lines of music ("voices") in a round.

Besides an Arduino board, the only hardware requirements are two resistors and a loudspeaker; no sensors are used in this version.

Essential ThreadKit

The main file of the **`muser`** demo, **`threads.cpp`**, shows how to use the essential elements of the **`ThreadKit`** library.

After including the necessary header files, declare the project's events:

`EVENTS (`*`event_1, event_2...`*`)`	Declare the names of events to be used in **`BEGIN_THREAD`** declarations and invocations of the **`broadcast`** function.

There are currently two built-in (predefined) events:

`NEVER`	This event never happens, and can be used as a convenient way to temporarily disable a thread for debugging.
`STARTUP`	This event happens once after the system is reset and **`user_setup`** runs (see below).

The **`user_setup`** function (analogous to Arduino **`setup`**) is called before any threads are started.

`void user_setup ()`	Insert any pre-thread initialization code in this function.

Create threads using the following three macros:

`THREAD (`*`name`*`)`	Declare a thread of the given ***`name`***. The thread's name is required by the **`THREADS`** macro which tells the library which threads to run, and may also be used in later versions for debugging and other purposes.
`BEGIN_THREAD (`*`event`*`)`	This must be the first line after the thread's opening brace. When the specified ***`event`*** is broadcast, the thread will start running. If it was already running, it will restart from the top. (This follows the Scratch specification.)
`END_THREAD`	This must be the last line before the thread's closing brace.

Within threads, you can use these macros at the top-level (but not in functions called from the thread):

`WAIT (`*`condition`*`)`	Block this thread until ***`condition`*** becomes **`true.`** The thread is always blocked (waiting) for at least one scheduler tick, regardless of ***`condition`***. (Scratch calls this *wait-until*.)
`WAIT_UNLESS (`*`condition`*`)`	Like **`WAIT`**, but don't yield control at all if the ***`condition`*** is already **`true.`** This variant is rarely necessary, and should not be used by novice programmers.
`SET_DELAY (`*`time`*`)`	Start a timed delay local to this thread. The units of ***`time`*** are milliseconds. Delay granularity is related to the scheduler frequency. See also **`WAIT_DELAY`**.
`IS_DELAY_OVER ()`	Returns **`true`** when the timed delay local to this thread expires.

Note: Variables that are local to threads will not be preserved across invocations of **`WAIT_`** or **`YIELD`** macros unless they are declared **`static`**. This is a consequence of conserving memory by not allocating a separate stack for each thread.

You can also use these shorthands:

WAIT_DELAY (*time*)	Block this thread for the specified time in milliseconds. The thread is always blocked for at least one scheduler tick. The resolution of the delay is determined by scheduler frequency. (Scheduler frequency is currently set to 100 Hz, for a delay resolution of 10 ms.)
WAIT_IMPATIENTLY (*condition*, *timeout*)	Block this thread until ***condition*** becomes **true** or ***timeout*** expires, whichever comes first. The thread is always blocked for at least one scheduler tick.

To trigger a thread, use the **broadcast** function:

broadcast (*event*)	Cause all the threads triggered by this ***event*** to start (or restart) running from their **BEGIN_THREAD**.

To create "safe" loops without making explicit calls to the **WAIT** macros and incurring their minimal delay of one scheduler tick:

YIELD ()	Yield control to the scheduler to avoid hanging the cooperative multithreading system. This is expected to be built-in to the Modkit implementations of looping constructs.

As an alternative to declared events, you can also use simple Boolean flag variables to communicate between threads. In a common idiom, one or more threads **WAIT**, while another thread signals an event with a "pulse" of the flag:

PULSE (*flag*)	Set the ***flag*** to **true** for one scheduler tick.

Note: **ThreadKit** scheduling macros are only meaningful when invoked at the top-level of a thread; you can't invoke them from a function called by the thread. This is the consequence of conserving memory by not allocating a separate stack for each thread.

Running Muser

To hear the output of muser, connect a small loudspeaker through two series resistors to Arduino digital outputs 9 (voice 1) and 11 (voice 2). With an 8-ohm speaker, 220-ohm resistors work well.

Five seconds after the **`STARTUP`** event (at reset), the program starts playing the first voice of the round in one thread. When the sixth note is about to be played by the first thread, it signals a second thread to begin. Five seconds after the second thread finishes, the entire process starts again.

Several parameters of the program by are intended to be set by means of a serial console. In this release, they are simply declared as variables in **`threads.cpp`**. The variables are **`song_delay`**, **`tempo_factor`**, and **`note_gap`**.