LiteOS Application Note
AN-103:
LiteOS Event Logging

Last updated: Jan 18 2008

This application note documents how to use the logging provided by LiteOS to record internal event traces for debugging, profiling, and application behavior analysis.

Purpose

This application note documents how to use the event logger that is built into LiteOS kernel to record traces of user applications and kernel behavior. The event logger is useful for post-experiment analysis, debugging, and profiling of user applications, in that it faithfully records all traces that the user is interested in. Using the event logger requires modifying the *eventlogger.h* file in the kernel and recompiles it. In this document, we explain the general rules we follow when we organize the *evnetlogger.h* file and its macro definitions.

Macro Definition Levels

Open the eventlogger.h file, you will observe the following macro definitions:

#define TRACE_ENABLE
#define TRACE_ENABLE_SYSCALLEVENT

These two macro definitions are the key control knobs for the event logging. If they are commented out, then the event logging functions will not be compiled into the kernel, because they are controlled using #ifdef macro in the kernel source code. On the other hand, if these definitions are effective, then the event logging code will be generated in the kernel image.

While the previous two macro definitions serve as the top-level macro, there are also many low-level control knobs also in the form of macro definitions. For example, the following definition:

#define TRACE_ENABLE_CONTEXTSWITCH

determines whether the context switch event will be logged or not by the kernel.

If such control knobs are enabled, each event is recorded as one byte in the event recording buffer, a chunk of memory whose size can be dynamically adjusted. If this buffer is full of events, it can be written into a file located in the root directory. The function:

void addTrace(uint8_t traceid)

is the key function that allows recording different types of events. This function is defined in the *eventlogger.c* file.

Another important category of logging is the system calls. In the *eventlogger.h* file, these macro definitions are organized as follows:

#define TRACE_ENABLE_SYSCALL_YIELDFUNCTION
#define TRACE_SYSCALL_YIELDFUNCTION   101

Notice that such definitions are organized in pairs. If the yield system call is logged, then the value 101 will be added into the *eventlogger* buffer. The trace could be retrieved at the end of experiments for post-experiment analysis.

Example to use the event logger

We use a very simple example, the "Hello, world" example to demonstrate the traces recorded by the event logger, and how such traces could provide very valuable insight on the behavior of the applications. The example we use is as follows. Note that, although we only use a very simple example here, the event logger is more useful for larger experiments where the behavior of the application could be highly complicated and even faulty. By examining the event traces, a user could locate potential sources of bugs and unexpected behavior. Further use in this direction will be documented in the future application notes.

```
int main()
{
int index;
  for (index = 0; index < 10; index++)
 {
   radioSend_string("Hello, world!\n");
   greenToggle();
   sleepThread(100);
 }
 return 0;
 }
```

By running this application and runs the event logger with only application specific events turned on (system calls and user application behavior), a user could easily get a trace file that is located under the root directory named *logtrace*. By copying this file back to PC, this file has a content like the following (Note the real experiment trace might differ slightly depending on which trace switches are turned on):

```
17
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
18
```

All such numbers correspond to individual events in the kernel, and simple analysis illustrates that the above trace

could be interpreted as follows:

//user thread starts
Create thread system call

(The following loop ten times)

//Radio_send function
Get radio mutex system call
Get current thread address system call
Get current radio data structure system call

//Mutex_lock function called in Radio_send
Get current thread address system call
Get current thread index system call

//Radio_send function
Radio send operation system call

//SleepThread function called in Radio_send
Get current thread system call
Yield system call

//Radio_send function
Reset radio state system call

//Mutex_unlock function
Mutex unlock system call

//Green_toggle function
Green toggle system call

//SleepThread function
Get current thread address system call
Yield system call

(The above loop ten times)
Destroy thread system call

As shown in this example, the event logger faithfully records the details of the logging procedure.

Conclusions

In this application note, we have briefly outlined the steps to use the event logger to reconstruct the program behavior in LiteOS applications.