Non-Invasive Execution History Tracing: a Diagnostic Aid for Embedded System Development

Use of a non-invasive execution tracing utility for tracing logic flow in the target environment during embedded systems development

Dean W. Neumann
Intel Corporation

1. Introduction

During embedded system development we are often faced with diagnosing and repairing software defects in a target environment without the benefit of a simulation environment or a rich toolset. Although we each have our own arsenal of tools and techniques which we draw upon to deal with such situations, adding additional tools to our arsenal is always beneficial. This article presents a public-domain utility which can be very useful in diagnosing a broad range of software errors, particularly in those environments where our primary development tools are not available.

There are many target environments in which our primary development tools may not be available. The target environment may be very resource limited, so that a debugger (even if available) cannot be loaded on to the target, or the target may not have a networking subsystem over which the debugger can communicate with the host. It may not have an I/O subsystem through which to output diagnostic messages. It may not have login or shell capabilities. The tools to generate core images and stack backtraces may similarly not be available. Sometimes all we have to work with is standalone firmware and a few bytes of dual-ported RAM or a JTAG port.

Even if our favorite tools are available on the target, there are a wide range of circumstances in which they cannot be used. Diagnosing an error which occurs intermittently in some interrupt handler under moderate or heavy load, for example, would prevent us from stepping through the software with a debugger, even if one was available. Similarly, determining which critical section of code is being intermittently interrupted by an asynchronously triggered function, such as a signal handler, would be difficult to do by setting watchpoints and executing the software under the debugger, even if one was available. Even if a core image or stack backtrace generating utility was available, it would not capture which signal handler interrupted the critical section, because the signal handler is no longer on the stack at the time the error is detected. Similar conditions prevent the use of the ubiquitous print statement. Adding print statements to software to localize errors is a popular brute force technique for following logic flow. However, print statements are computationally expensive, and the same conditions which prevent the use of a debugger frequently also prevent the use of print statements. Print statements may not be available in kernel space or in routines which must not block, or the I/O subsystem may not even be available in the target. Even when print statements are an option, they are usually a poor choice, because either they provide too coarse a resolution to adequately determine logic flow in nontrivial asynchronous software, or they involve modifying the software to insert numerous print statements in the hope that they will be sufficient. In this case, the problem of I/O buffering vs. CPU waiting must be dealt with.

The utility presented here allows software under test to be procedurally instrumented at compile time, so that it collects a trace of its own execution at runtime. The trace can then be analyzed to determine the logic flow of the software before and during error occurrence.

2. A proposed solution

The solution presented, called “Xhist” (for “execution history”) involves instrumenting the software under test at compile time, so that every source statement has the corresponding filename and line number written to a circular buffer in RAM as it is executed, resulting in a history of the last N statements executed. Then, either at program termination or upon demand, the circular buffer can be analyzed (either directly in memory, or if I/O is possible it can be exported from RAM to any I/O device) so that the last N statements executed can be reported to the developer.

The solution also includes a library which can be optionally linked into the software under test, providing a default function for exporting the circular buffer to a file or socket descriptor, and a post-processing utility which can read the binary data exported by the library function and generate an ordered, human-readable listing of the circular buffer up to the last statement executed.

By reviewing the execution history of the software up to the point of failure, the developer can trace the program’s logic flow, including asynchronously invoked functions such as interrupt handlers or signal handlers, callback functions, etc. The developer can see which logic executed that was not expected (such as infinite loops, incorrect loop counts, or `else` clauses) and one can also see the *absence* of logic that was expected to execute (indicating that a timer did not fire, or an interrupt was not received, etc). These are useful in diagnosing many classes of software (and sometimes hardware) defects.

The xhist_instrument utility

Execution history tracing begins by instrumenting the software prior to compilation. The instrumentation is performed by the `xhist_instrument` utility, which is a simple lex program that generates a lexical scanner for C or C++ source files, and translates each semicolon terminating an executable statement into a macro call. The macro call is then expanded by the preprocessor into a memcpy of the source filename and line number into the circular buffer, after the execution of each statement. The macro is defined in an include file “`xhist.h`” which is included into the software under test. This is most easily done by nesting the inclusion of xhist.h within a common include file already included by the source files to be compiled. The inclusion of xhist.h can in fact be left in the software permanently, since it is surrounded with “`#ifdef XHIST`” preprocessor conditionals and therefore has no effect unless enabled by defining the XHIST preprocessor directive.

The instrumented software is then compiled using the same compiler and flags that would normally be used to compile the software for the target environment. The `xhist_instrument` utility may therefore be used with a native compiler or cross-compiler, and the instrumented source code is still “recognizable” when viewed in a graphical development environment or symbolic debugger. No cross-compiler or additional tools are required -- just a header file defining the macro, the size of the circular buffer, etc.

The lexical scanner is shown in Figure 1. Sample instrumented code produced by the xhist_instrument utility is shown in Figure 2. The example shows how the utility has appended a macro call to each executable statement, and also shows how sections of code can be excluded from instrumentation by surrounding them with comments containing directives which the utility recognizes.

The macro which is appended to each statement incurs no function call overhead except for one `memcpy()` to copy the filename into the circular buffer, and even that could be replaced by an inline loop in either C or assembler to save the cost of the function invocation, but at the expense of code size. Since the number of bytes in the filename to be copied to the circular buffer is small and bounded, and the cost of a function invocation can be determined for the specific target environment, a determination can be made by the developer as to whether it is preferable to incur the function invocation overhead or the additional code size of inline loops. It is also possible to extend the lexical scanner to recognize inline assembler or compiler extensions for specific target environments, or to detect additional directives and append a different macro name, so that inline loops are used only in critical sections, with the `memcpy()` invocation being used in non-critical sections.

The xhist library

The instrumented software will include a header file `xhist.h` which defines the size of the circular buffer and the data structure that makes up each element. The size can be configured based on the resources available in the target environment and the length of history needed, by altering the constant defined in the header file. If the target environment offers the ability to access the circular buffer externally even after program termination or after the program ceases to respond to external events, (if the buffer is held in dual-ported RAM or shared memory for example), no exporting function is required, and the xhist library need not be linked into the software under test. However, if, as is usually the case, it is desired that the circular buffer be exported from its memory location upon program termination or under user or program control, the xhist library can either be compiled directly into the software under test as is, or it can be precompiled separately for the target environment and made available to all the developers on the team to simply link into the software under test.

The xhist library defines a function `xhist_write()`, which writes the circular buffer as unformatted binary data to an arbitrary open file or socket descriptor. The `xhist_write()` function has the appropriate signature to allow it to be installed as both a signal handler via the POSIX `signal()` function and as a termination handler via the XPG4 `atexit()` function, so that the circular buffer will be automatically exported upon receipt of a specified signal or upon program termination. It can also be invoked upon demand, such as when the program reaches a particular state as determined by polling a memory location, waiting on a semaphore, etc.

To be installable as both a signal handler and a termination handler, `xhist_write()` accepts no function arguments. So the xhist library also includes an encapsulation function `xhist_logdev()` to set the file descriptor that the `xhist_write()` function should write to. Figure 3 shows how the software under test registers the `xhist_write()` function as both a handler for SIGUSR1 and as a termination handler. This initialization is enclosed in an

`#ifdef XHIST` preprocessor directive so that it need not be removed from the code. In fact, since the header file xhist.h is also internally enclosed by the same preprocessor directive, its inclusion and the registration of `xhist_logdev(fileno(stderr))` or other appropriate descriptor at the beginning of each program can be made a project coding standard, so that every program is inherently traceable just by defining the XHIST flag.

The xhist_report utility

If the `xhist_write()` function was used to export the circular buffer as binary data, the data will need to be formatted to be human readable. The `xhist_report` utility is an additional program that may be used for this purpose. It reads the data stream (as written from the circular buffer) on its standard input and `printf()`s it to its standard output, in order from oldest to latest statement executed. The data stream can come from a previously written file, or directly from the target system over a socket or a serial port. The `xhist_report` utility performs byte swapping if necessary, so it need not run on the same architecture as the software under test. An example of a formatted execution history is shown in Figure 4.

3. A Practical Example

Consider the common example of an embedded program consisting of a state machine which registers interrupt handlers, starts timers, registers callback functions, and then enters an event loop responding to synchronous and asynchronous events as they occur. Having downloaded the program to the target environment and started it's execution, the program behaves correctly for an extended period of time, but then stops responding to events. The mean time to failure appears to be random. Xhist can help determine what the program did just prior to failure. Figure 4 shows an excerpt from an execution history from just sucha n example. The filenames in the listing have been chosen so that it would be intuitive to a broad audience what these lines of code represent, but in a real project this is not necessary, since the developers would recognize their own filenames and would certainly have the code at hand so that they could determine exactly which lines of code are represented in the listing.

In Figure 4 we see a recurring pattern in which the file `StateMachine.c` invokes a function from the file `CritSection.c` (`CritSection.c`:26 and `CritSection.c`:27) as it processes synchronous events, and invokes functions from the file `EventHandlers.c` as it processes asynchronous events. We see that `EventHandlers.c` also invokes the same functions of `CritSection.c`, namely `CritSection.c`:26 and `CritSection.c`:27. This pattern represents the program's expected behaviour. However, we can clearly see the abnormality in the pattern which caused the failure. At line 39 of the listing, we see `StateMachine.c` invoking `CritSection.c`:26, but then `CritSection.c`:27 isn't executed until 7 lines later. Line 41 shows that the critical section was interrupted by another event, and that event handler re-invoked the critical section. Our program failure is the result of incorrect mutex locking in the ciritical section, or incorrect interrupt queueing in the event handlers.

4. Summary

Xhist is a diagnostic aid that provides programs with the ability to produce an ordered list of the source statements executed by the software under test before and during the point of

failure. It provides very fine grained resolution (each source statement), while avoiding the need for I/O during program execution, and without the need to manually add debug statements to the software under test. Xhist is a component of the eXtensible Management Toolkit (XMT) and is freely available under the terms of the GNU General Public Licence, version 2. XMT and its various packages has been used by various companies in the development of unified messaging, voice over IP, and paging switches for several years.

Intel's Networking Software Division assists customers with the development of embedded software. We use Xhist as an integrated component of our standard Makefiles and cross-platform build tools, allowing us to "make xhist" in any project or target environment with minimal effort and change to the software under test. Adding this and other diagnostic tools to our developers' arsenal helps us remain competitive, as it can for you.

Xhist is available for download under the terms of the GNU General Public License, from http://www.trillium.com/professional-services/clients/dload1000000/xmt_xhist.tgz

```
%{ /* DO NOT REMOVE THIS LINE */

#include <stdio.h>
#define TRUE    1
#define FALSE   0
#define TRACE   "_XH"

char    in_rtn     = FALSE; /* TRUE if inside a routine  */
char    debug      = FALSE; /* TRUE if debugging scanner */
char    instrument = TRUE;  /* TRUE if instrumenting     */

%}

word    [a-zA-Z0-9_]+
operator        [-+<>=!\^\(]

%%
^[ \t]*\/\*[ \t]*xhist[ \t]+debug[ \t]+TRUE[ \t]*\*\/\n {
/* "xhist debug TRUE" inside a comment enables tracing   */
    debug = TRUE;
    printf("%s%s",  (debug ? "<DEBUG ON>" : ""), yytext);
}

^[ \t]*\/\*[ \t]*xhist[ \t]+debug[ \t]+FALSE[ \t]*\*\/\n        {
/* "xhist debug FALSE" inside a comment disables tracing */
    printf("%s%s",  (debug ? "<DEBUG OFF>" : ""), yytext);
    debug = FALSE;
}

^[ \t]*\/\*[ \t]*xhist[ \t]+instrument[ \t]+TRUE[ \t]*\*\/\n    {
/* "xhist instrument TRUE" inside a comment
    enables instrumentation */
    instrument = TRUE;
    printf("%s%s",  (debug ? "<INSTRUMENTATION ON>" : ""),
           yytext );
}

^[ \t]*\/\*[ \t]*xhist[ \t]+instrument[ \t]+FALSE[ \t]*\*\/\n   {
/* "xhist instrument FALSE" inside a comment
    disables instrumentation */
    instrument = FALSE;
    printf("%s%s",  (debug ? "<INSTRUMENTATION OFF>" :""),
           yytext );
}

^\{[ \t]*$      {
/*  an opening brace in column 1
    indicates a procedure start */
    in_rtn = TRUE;
    printf("%s%s",  (debug ? "<FUNC START>" : ""), yytext);
}

^\}             {
/*  a closing brace in column 1 indicates a procedure end*/
    in_rtn = FALSE;
    printf("%s%s",  (debug ? "<FUNC END>" : ""), yytext);
}

^[ \t]*{word}\**[ \t]+\(?\**{word}.*\n  {
    /*  an identifier followed by whitespace then another
        identifier indicates a declaration */
        printf("%s%s",  (debug ? "<DECL>" : ""), yytext);
    }

    .*[ \t]*for[ \t]*\(.*\;.*$      {
    /* "for" keyword indicates a for statement */
        printf("%s%s",  (debug ? "<FOR STMT>" : ""), yytext);
    }

    [ \t]*return[ \t\(]+.*\)*\;     {
    /* "return" keyword indicates a return statement */
        if ( in_rtn )
        {
            printf("%s{%s;%s}",  (debug ? "<RETURN STMT>" :""),
                  (instrument ? TRACE : ""), yytext);
        }
        else
        {
            printf("%s%s",
                   (debug ? "<RETURN STMT OUTSIDE FUNC>" : ""),
                    yytext);
        }
    }

    .*{operator}.*\;        {
    /*  a line containing an operator and terminating
        with a semicolon indicates a statement */
        if ( in_rtn )
        {
            printf("%s%s%s",
                   (debug ? "<STMT>" : ""), yytext,
                   (instrument ? TRACE : ""));
        }
        else
        {
            printf("%s%s", (debug ?
                   "<STMT OUTSIDE FUNC>" : ""), yytext);
        }
    }

    \n  {
    /*  a carriage return is simply echo'ed */
        printf("%s", yytext);
    }

    .   {
    /*  a nything else is simply echo'ed */
        printf("%s", yytext);
    }

    %%

    yywrap()
    {
        return( 1 );
    }
```

Figure 1: Xhist_instrument lexical scanner

```
void    *cp_worker_main( CP_THREAD *th )
{
    CP_THREADPOOL       *tp;
    CP_JOB              *job;
    sigset_t            sigset;
    void                cp_worker_cancel();


    /*
     *  setup thread cancellation & sigmask attributes
     */

    tp = thread_threadpool(th);_XH
    sigfillset(&sigset);_XH
    (void) pthread_sigmask(SIG_BLOCK, &sigset, (sigset_t *)NULL);_XH
    (void) pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);_XH
    (void) pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);_XH
    pthread_cleanup_push(cp_worker_cancel, th);_XH


    /*
     *  enter main loop.
     *  wait for a job & then process it.  Repeat forever.
     *  Mutexes are not required on the thread control variables because
     *  the boss thread only modifies the variables when the worker is idle
     *  and the worker only modifies them when it's active.
     */

    while(1)
    {
        pthread_cond_wait( threadpool_jobq_cv(tp), threadpool_jobq_mx(tp) );_XH
        if ( (job = threadpool_dequeue_first(tp)) != (CP_JOB *) NULL )
        {
            thread_job(th) = job;_XH
            job_thread(job) = th;_XH
            thread_state(th) = CP_THREADSTATE_ACTIVE;_XH
            cp_build_message(job);_XH
            thread_state(th) = CP_THREADSTATE_COMPLETED;_XH
            pthread_testcancel();_XH
        }
    }

    /*
     *  we never reach the following statement, but some compilers require
     *  matching pthread_cleanup_push() & pthread_cleanup_pop() calls.
     */

    /* NOTREACHED */
    pthread_cleanup_pop(TRUE);_XH
   {_XH;    return((void *)NULL);}
}
```

Figure 2: Sample instrumented source code

```
int main( int argc, char* argv[] )
{

#ifdef XHIST
    int fd;
    extern int atexit();

    /*
     *  if we can open an xhist logfile, set the log device to the
     *  file descriptor returned, then install the xhist_write
     *  function to be called upon termination, and upon receipt of
     *  the named signal.
     */

    if ((fd=open(XHIST_LOGFILE, O_RDWR|O_CREAT, 0644)) < 0)
    {
         perror(XHIST_LOGFILE);
         exit(1);
    }
    xhist_logdev(fd);
    atexit(xhist_write);
    signal(SIGUSR1, xhist_write);
#endif
```

Figure 3: Registration of the xhist_write function

```
 1 StateMachine.c:93
 2 StateMachine.c:68
 3 StateMachine.c:49
 4 StateMachine.c:48
 5 StateMachine.c:98
 6 CritSection.c:26
 7 CritSection.c:27
 8 StateMachine.c:99
 9 StateMachine.c:100
10 StateMachine.c:71
11 StateMachine.c:49
12 EventHandlers.c:18
13 CritSection.c:26
14 CritSection.c:27
15 EventHandlers.c:19
16 EventHandlers.c:20
17 EventHandlers.c:21
18 StateMachine.c:48
19 StateMachine.c:112
20 CritSection.c:26
21 EventHandlers.c:18
22 CritSection.c:26
23 CritSection.c:27
24 EventHandlers.c:19
25 EventHandlers.c:20
26 EventHandlers.c:21
27 CritSection.c:27
28 StateMachine.c:113
29 StateMachine.c:114
30 StateMachine.c:77
31 StateMachine.c:49
32 EventHandlers.c:18
33 CritSection.c:26
34 CritSection.c:27
35 EventHandlers.c:19
36 EventHandlers.c:20
37 EventHandlers.c:21
38 StateMachine.c:48
39 StateMachine.c:98
40 CritSection.c:26
41 EventHandlers.c:18
42 CritSection.c:26
43 CritSection.c:27
44 EventHandlers.c:19
45 EventHandlers.c:20
46 EventHandlers.c:21
47 CritSection.c:27
48 StateMachine.c:99
49 StateMachine.c:100
50 StateMachine.c:71
```

Figure 4: Xhist execution trace (excerpt)