Linux OS-Aware Debugging

Overview

Today Linux is the fastest growing embedded operating system in the world. Bringing Linux up on an embedded target and debugging it and associated applications present new challenges for embedded systems programmers. These challenges, in turn, must be addressed by the designers and suppliers of tools for these programmers.

It is difficult to port to a new platform without some method of accessing the system boot information console and the login console. On many headless targets there is either no good candidate for this functionality or such candidate is not available at boot time. For example, cell phones simply do not have serial interfaces. In addition, the MMU-based memory model makes classic halt-mode debugging difficult or even impossible. Because all tasks share the same virtual address in Linux, classic breakpoint methods need modification. Debug register breakpoints (hardware breakpoints) are of little use.

This application note describes a debug topology where halt mode run control is combined with run mode debugging and TCP/IP-based communications to effectively solve all of these issues. This method provides a test environment console that uses virtually no production resources. In addition, it combines the best features of halt-mode and run-mode debugging and offers the programmer a feature-rich and effective way to get an embedded Linux-based project completed on time and fully tested.

Linux in the Embedded World

Embedded designs are changing in nature and requirements. In the past, embedded usually referred to relatively small computer systems contained in a hardware product that showed almost no characteristics of a computer. This hardware product contained tiny amounts of memory and, therefore, small and very efficient programs. These products usually had neither the means nor the requirement to communicate with other programmed products. If these machines contained operating systems at all, they were usually lean and not very feature rich. Small and fast were the key drivers for these systems.

Today's embedded projects are becoming more and more sophisticated, both in terms of features and underlying hardware. Most embedded systems include some type of connectivity, such as Ethernet or USB. Cheaper and faster processors and memory are increasingly becoming available for embedded systems. Practically all of today's smart-phones accepted downloaded programs. Along with more software and connectivity comes the requirement for remote configuration and management.

With all of these software-based features being added to embedded products, software reuse is more necessary than ever. Developers must acquire operating systems, protocol stacks, http servers, remote management and communication software and many other software components in order to produce the required features. Writing these components can produce noncompetitive development costs; these components must be selected from those in general use to be competitive. Fortunately, many of these components exist in the public domain and in the open source community. In order to deploy the software components required, the design must be based on an operating system on which these components can run. That tends to limit the field to Linux derivatives.

Debug/Deployment Challenges

In order to use Linux, embedded engineers must first port it to their hardware. They then must write and debug any custom drivers required for their application as well as write and debug the boot-up code. The final stage is to write and debug the applications that will run on Linux. This process offers several tool challenges that do not exist when using more classic operating systems such as Wind River® VxWorks® or other flat address environments.

Porting Linux to a new target requires a board support package (BSP) or boot code just like any other solution. Once the boot code instantiates and attempts to launch the kernel, several new requirements surface. Linux is a large body of code; the first order of debugging a new port is to observe the kernel process while booting to determine what has succeeded, what has failed, and where the process stopped. One common way of monitoring this process is to observe the system console messages. This is obvious, but not necessarily easy. Many embedded applications have no UART or other common port on which to attach a console. Even if the final product does have such a port, it may not be functional until new drivers are written and debugged. If this is the case, debug of the driver is going to be a serious challenge since the kernel may still have problems. Getting that first kernel message may be difficult.

Most JTAG run-control probes use a debug state called halt mode. In this state the processor is either physically stopped or redirected to special software that makes it look like it is physically stopped. The firmware engineer then examines and makes changes to the system in an entirely static condition. This is the view that is offered by most embedded debugging tools. For Linux application debug, this may present problems.

Once the kernel is booting successfully, the process of introducing and debugging the embedded application begins. Here another real challenge is encountered. Unlike operating systems that use a flat memory model, Linux dynamically maps all applications to the same virtual address space. This renders debug register-based breakpoints virtually worthless in debugging application code; in most architectures, debug register-based breakpoints (sometimes called hardware breakpoints), monitor the virtual address. This is true for all ARM®-architecture processors. Because these comparators are attached to the virtual address bus (IVA or DVA), they will detect a match to a particular virtual address without regard to the physical address currently mapped. This will cause the same address in any application (say 0x8330) to cause a trigger. This means halt-mode debugging in the application may not be viable. Software breakpoints are clearly the answer. In order to use them, the halt-mode debugger must know where the code is physically at all times and be aware of any reloads caused by the kernel. Address aliasing between user tasks also presents a new kind of challenge for locating and displaying the source code related to the program being debugged. In addition to the code address aliasing (physical versus virtual), another factor is the need to use a console-based login to control and observe tasks and applications. This may present the same problems as the initial console.

Some of these challenges may be met via kernel-aware halt-mode debuggers. These debug solutions need to be kept in sync with a rapidly changing code base for Linux.

A Combined Paradigm

A combined approach can be used to solve the debug challenges encountered when working with embedded Linux. ASSET InterTech, Inc. provides tools for ARM-based and Intel processor markets that are based on a combined paradigm. This paradigm has two parts. The first is a debug port connection to the processor that provides a hardware port that is independent of target system peripherals. The second part is a debug kernel that runs on top of Linux that communicates to the remote hosted debug session via this virtual port. This virtual port also supports the console facilities.

The virtual communications port is based on an ADB server and the client running in SoucePoint. SourcePoint provides both a console connection and debug communications through the Arium run-control probe. (See Figure 1.)

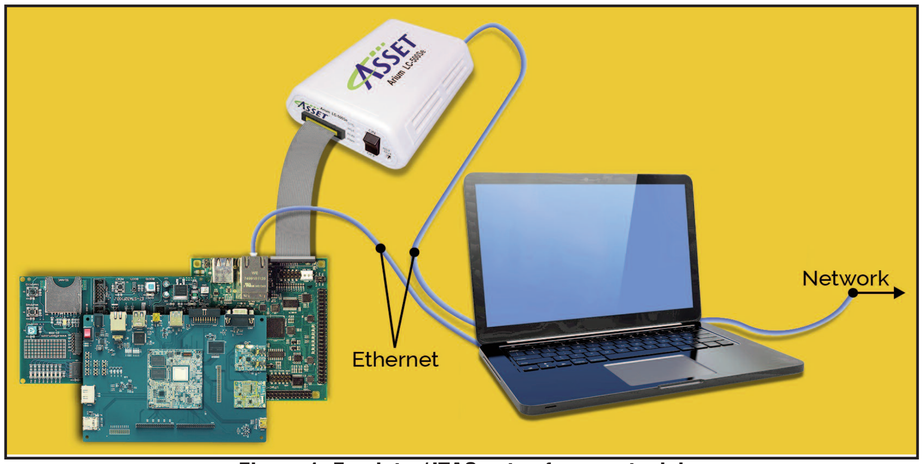


Figure 1: Emulator/JTAG setup for remote debug

In this way, console is available from power-up without any other hardware dependencies except the processor core and the JTAG debug port. This port is almost always used during boot loading. In this way, the very first messages produced by the kernel are available to the embedded software engineer. (See Figure 2.)

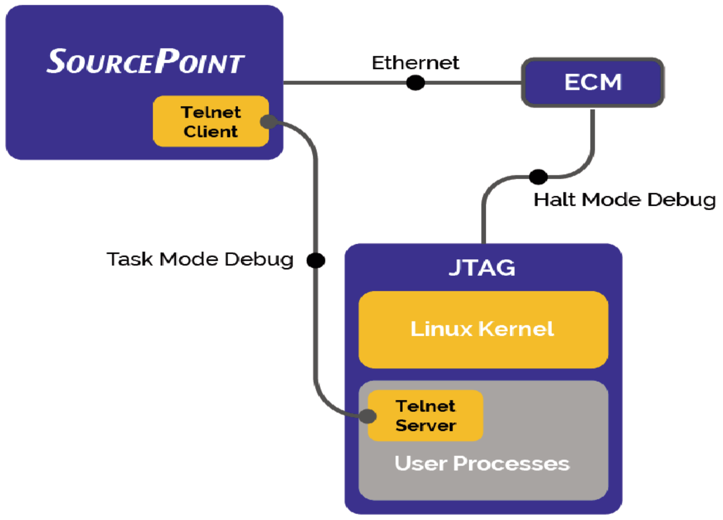
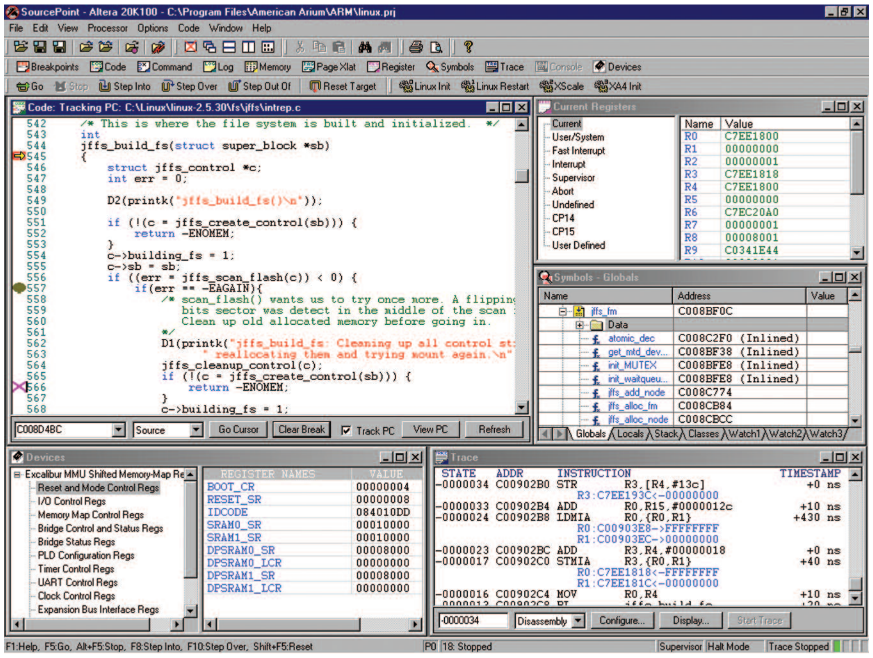


Figure 2. SourcePoint™, the ASSET debugger, gains access to the Linux kernel via the ADB Server.

Once the kernel is up, all user space software debugging and program control is done via a debug agent running as a user process under Linux. The agent uses standard Linux APIs to control and interrogate user space programs. In this way, the debugger can more easily stay in sync with subsequent kernel versions. The agent runs during the entire process so all of the usual system information is available, either through Linux utilities (such as ps) or directly from the remotely hosted debugger. The embedded engineer has his or her choice of tools and methods for each problem.

All debugging can be accomplished via the company's flagship SourcePoint debugger. (See Figure 3.) When using SourcePoint, the engineer may switch between task- and halt-mode debugging at any time; the interface directly into the kernel is visible in its entirety and rendered at source level with all symbols handled. Breakpoints can be set in


SourcePoint - Altera 20K100 - C:\Program Files\American Arium\ARM\linux.prj
File Edit View Processor Options Code Window Help
Breakpoints Code Command Log Memory Page Xlat Register Symbols Trace Console Devices
Go Stop Step Into Step Over Step Out Of Reset Target Linux Init Linux Restart XScale XA4 Init
Code: Tracking PC: C:\Linux\linux-2.5.30\fs\jffs\intrep.c
C008D4BC Source Go Cursor Clear Break Track PC View PC Refresh
Current Registers
Current User/System Fast Interrupt Interrupt Supervisor Abort Undefined CP14 CP15 User Defined
Symbols - Globals
Globals Locals Stack Classes Watch1 Watch2 Watch3
Devices
Trace
-0000034 Disassembly Configure... Display... Start Trace
F1:Help, F5:Go, Alt+F5:Stop, F8:Step Into, F10:Step Over, Shift+F5:Reset
P0 18: Stopped Supervisor Halt Mode Trace Stopped

```
/* This is where the file system is built and initialized.  */
int
jffs_build_fs(struct super_block *sb)
{
    struct jffs_control *c;
    int err = 0;

    D2(printk("jffs_build_fs()\n"));

    if (!(c = jffs_create_control(sb))) {
        return -ENOMEM;
    }
    c->building_fs = 1;
    c->sb = sb;
    if ((err = jffs_scan_flash(c)) < 0) {
        if(err == -EAGAIN){
            /* scan_flash() wants us to try once more. A flipping
               bits sector was detect in the middle of the scan
               Clean up old allocated memory before going in.
            */
            D1(printk("jffs_build_fs: Cleaning up all control st
                   " reallocating them and trying mount again.\n"
            jffs_cleanup_control(c);
            if (!(c = jffs_create_control(sb))) {
                return -ENOMEM;
            }
            c->building_fs = 1;
```

Name	Value
R0	C7EE1800
R1	00000000
R2	00000001
R3	C7EE1818
R4	C7EE1800
R5	00000000
R6	C7EC20A0
R7	00000001
R8	00008001
R9	C0341E44

Name	Address	Value
jffs_fm	C008BF0C	
Data		
atomic_dec	C008C2F0 (Inlined)	
get_mtd_dev...	C008BF38 (Inlined)	
init_MUTEX	C008BFE8 (Inlined)	
init_waitqueu...	C008BFE8 (Inlined)	
jffs_add_node	C008C774	
jffs_alloc_fm	C008CB84	
jffs_alloc_node	C008CBCC	

Excalibur MMU Shifted Memory-Map Re
- Reset and Mode Control Regs
- I/O Control Regs
- Memory Map Control Regs
- Bridge Control and Status Regs
- Bridge Status Regs
- PLD Configuration Regs
- Timer Control Regs
- UART Control Regs
- Clock Control Regs
- Expansion Bus Interface Regs

REGISTER NAMES	VALUE
BOOT_CR	00000004
RESET_SR	00000008
IDCODE	084010DD
SRAM0_SR	00010000
SRAM1_SR	00010000
DPSRAM0_SR	00008000
DPSRAM0_LCR	00000000
DPSRAM1_SR	00008000
DPSRAM1_LCR	00000000

```
STATE    ADDR     INSTRUCTION                          TIMESTAMP
-0000034 C00902B0 STR        R3,[R4,#13c]                  +0 ns
                    R3:C7EE193C<-00000000
-0000033 C00902B4 ADD        R0,R15,#0000012c             +10 ns
-0000024 C00902B8 LDMIA      R0,{R0,R1}                  +430 ns
                    R0:C00903E8->FFFFFFFF
                    R1:C00903EC->00000000
-0000023 C00902BC ADD        R3,R4,#00000018               +0 ns
-0000017 C00902C0 STMIA      R3,{R0,R1}                   +40 ns
                    R0:C7EE1818<-FFFFFFFF
                    R1:C7EE181C<-00000000
-0000016 C00902C4 MOV        R0,R4                        +10 ns
```

Figure 3: SourcePoint on a Linux platform

each task that break only when that task reaches that point. All of these features can be greatly enhanced with real time trace capabilities in the same tools.

Task-mode Features

Task-mode debugging offers features that are not available or are, at best, marginal in halt mode. These features include:

- Full process observability
- Full console support independent of target peripherals
- Target file system write (for such things as application introduction)
- Remotely hosted debugger using only the JTAG connection
- Seamless connection to halt mode
- Real-time trace

These features are in addition to the expected ones such as:

- Single stepping
- Running to breakpoints
- Viewing programs
- Viewing and changing data objects in their natural format

Consequently, in task mode, programming engineers can debug their applications with full knowledge of them and other processes. They can examine the state of each process and understand why some are active and others are idle. The program and all of its data can be displayed using all available source level information.

Seamless Task Mode/Halt Mode - The Best of Both Worlds

While task-mode debug is the clear answer for application debug in Linux, there are times when halt mode is still the answer. If the engineer is tracing a call down into the kernel, he or she will want to stop the kernel and look at what is going on. This follows through to kernel drivers. These areas are all fully visible only in halt mode (with the processor stopped). SourcePoint allows setting of hardware breakpoints that cause the processor to enter halt mode during a task-mode debug session. In this way, a problem or code path can be followed in its entirety. ARM's Embedded Trace Macrocell (ETM) allows full sequential breakpointing to be used to enter halt mode.

In addition to debugging across the application/kernel boundaries, halt mode is very powerful in scripted board initialization. In this way, the kernel can be launched and debugged long before the boot code is written or functional. Kernel and application work can proceed while the boot code is being written and debugged. This produces faster times to market.

Key features include:

- Kernel download
- Programming flash memory
- Real-time trace (including precise time stamping)
- Profiling

Conclusion

Using Linux in the embedded world will be much easier with the introduction of some new tools that tackle the tough obstacles of this job. (See Figure 4.) The biggest obstacles can be classified into two areas. These are (1) the need for a virtual port into the target hardware system, and (2) the need to get clear display and control of applications in spite of the address aliasing issues. SourcePoint has the communications facility that can supply this virtual port.

Figure 4: The Arium LC-500Se run-control probe is designed for Linux support