Memory Debugging of Virtual Prototypes with TLM 2.0

George F. Frazier
Cadence Design Systems, Inc.
georgef@cadence.com

Qizhang Chao
Cadence Design Systems, Inc.
qzc@cadence.com

Neeti Bhatnagar
Cadence Design Systems, Inc.
neeti@cadence.com

Kathy Lang
Cadence Design Systems, Inc.
lang@cadence.com

***Abstract*— Memories are commonly modeled as TLM 2.0 components in SystemC-based virtual prototypes. The role of memory models and memory subsystems in a virtual prototype is multi-fold. Memory models are often at the center of debugging activities for both hardware and embedded software.**

Because the TLM function transport_dbg allows inspection of target values without side effect, it is possible to create generic memory debugging tools for any design that models memory as a TLM 2.0 component. This work examines the aspects of the TLM 2.0 standard that support memory debugging, in particular the transport_dbg function.

We demonstrate with real debug problems encountered in the development of a virtual prototype that runs an embedded OS and other embedded software. We use the example to show how TLM 2.0 supported memory debugging can be used as part of an arsenal of tools to investigate debug problems familiar to virtual prototype designers.

Keywords: TLM, System Level Design, Virtual Prototypes, Virtual Platforms, SystemC, Memory Modeling, Hardware Software Co-design, Android.

I. INTRODUCTION

Memories are commonly modeled as TLM 2.0[1] components in SystemC-based virtual prototypes (VPs). The role and use of memory models and memory subsystems in a virtual prototype is multi-fold. The program memory, data memory, and stack need not be stored in a single memory model – they can be split between multiple memory models. Alternately, they can be stored in a single memory model but implemented with sparse memory or other advanced memory modeling techniques that are often predicated by performance: loosely-timed (LT) models have to be implemented with the utmost efficiency. Memory models are often at the center of debugging activities for both hardware and embedded software.

Because the TLM function transport_dbg allows inspection of target values without side effect, it is possible to create generic memory debugging tools[2,3] for any design that models memory as a TLM 2.0 component. Such frameworks do not require the user to do any hand-coding other than to specify the system memory map (either through proprietary methods or through a standard such as IP-XACT). Such a framework can provide a system-level view of the memory-mapped target devices of a virtual prototype, interactive memory value viewing, memory cell or vector update, software disassembly display, and memory value tracing - all in a design-agnostic fashion.

This work begins by examining the aspects of the TLM 2.0 framework that support memory debugging, in particular the use of the standard transport functions to inspect and modify memory values, and how the use of TLM dmi affects visibility into memories (for example dmi supports access of different memory blocks/pointers from a single socket). Dmi can make a model run very fast, but it can also make memory debugging a difficult challenge.

We then demonstrate how the framework was used to solve difficult debug challenges during development of a virtual prototype that runs an embedded OS and other embedded software. The example includes dual processors – an ARM Cortex-A9 processor model and an ARM cortex M3 processor model - and a TLM-modeled RAM that boots an Android image. The Android image is ported to a SystemC-based virtual prototype that includes a collection of peripherals such as audio, battery, tty Ethernet, loader, and the Android skin all implemented in SystemC.

Using the development of the Android virtual prototype, we show how the approach can be an essential aid in finding and fixing two classes of problems familiar to anyone that works with virtual prototypes:

- Hardware peripheral design or Processor configuration errors that are discovered during early software development and require memory model access at the SystemC debug level. Often the hardware errors are discovered during the embedded OS port and involve dual hardware/software debug challenges.
- "Bare metal" device driver logic and endianness errors that are very difficult to debug.

The solutions require different debugging abstractions and approaches, but the capability of the TLM 2.0 standard to support target value introspection and modification allows the construction of powerful, integrated debug environments that operate across multiple levels of hardware and software development abstractions.

II. MEMORIES IN TLM 2.0

A. *Modeling Memory with TLM 2.0*

VPs of systems on a chip (SOC) generally contain hardware modules that access a shared memory sub-system through a memory interconnect[4]. At the same time, application software can be running on the chip, or in the case of multi-core designs, multiple chips, which results in software-originated memory access requests. A common memory sub-system implemented in SystemC/TLM-2.0 can be shared by both the hardware models of the VP and the software running on the system. A challenging goal of VPs is to efficiently model this shared access, either at the architectural or AT-level of TLM 2.0 or the higher level of abstraction provided by the loosely-timed style of TLM 2.0 modeling. Loosely-timed models provide less accuracy but typically much greater throughput (cycles per second) than AT models which provide greater accuracy at the cost of performance.

There are several ways memory subsystems are used in VPs, and in fact there is often more than one memory model in the system. Besides memory-mapped registers used for hardware communication, software requires program memory, data memory, and stack storage, which can be isolated in a single memory or split across multiple memory models. Whether the implementation uses sparse memories[5] or other advanced memory modeling techniques, the ability to inspect memory values plays a critical role in debugging either the hardware design or the embedded software running on the system – and often in the case of VPs – both at the same time.

Because of the generic nature of the TLM generic payload and the standardization of well-defined access functions for targets, it is possible to construct tools that allow inspection and modification of the memory subsystems of a VP without explicit hand-coding or modification of the SystemC code used to implement the memories (the only requirement is a "complete" implementation of the memory, i.e. appropriate dmi and transport_dbg functions must be correctly implemented for each model). The target sockets of the hardware components accept transactions to model the read and write operations to the memory of the component. This allows creation of tool features that allow memory debug operations such as interactive memory state inspection, interactive memory cell modification, and tools, such as memory viewers, that aggregate these operations in a debugger or IDE.

System memory maps are not a part of either the SystemC or TLM standards, but are almost always part of a VP and are important to memory debugging. The system memory map is used to interpret the address field of the generic payload that enables tools to provide sophisticated debug information that is either target or system address aware.

B. *TLM 2.0 functions for inspecting target values*

Memories are implemented as targets in TLM 2.0, and so in principle all of the TLM transport functions and APIs can be used to access memory values. The access value of the generic payload field can be set to either read or write according to the desired operation.

However b_transport, nb_transport_fw, and nb_transport_bw functions have side effects (Table 1). The TLM function transport_dbg, however, is an interface designed to allow the initiator to read or write memory in the target without either side effects in the system or the passage of time. The transport_dbg function can be used by memory debugging tools as the basis of memory inspection and modification.

The transport debug interface is a singleton (it does not have separate blocking and non-blocking versions) that uses the forward path exclusively. Sometimes TLM designs will supply "do-nothing" implementations of transport_dbg, but for memory debugging to work, correct implementations are required (Appendix A shows am example of a simple transport_dbg function that might be used for a TLM 2.0 wrapped VP).

```
unsigned int
transport_dbg(tlm::tlm_generic_payload
&trans){}
```

The transport_dbg method takes only the generic payload argument rather than the additional arguments of the transport interface, and indeed it operates on a restricted set of the generic payload fields: the command, address, data pointer, and the data length. The goal of the function is to copy as many bytes as possible and to return the number of actual bytes copied. A debug environment can then query the target by calling transport_dbg somewhere in appropriate initiator:

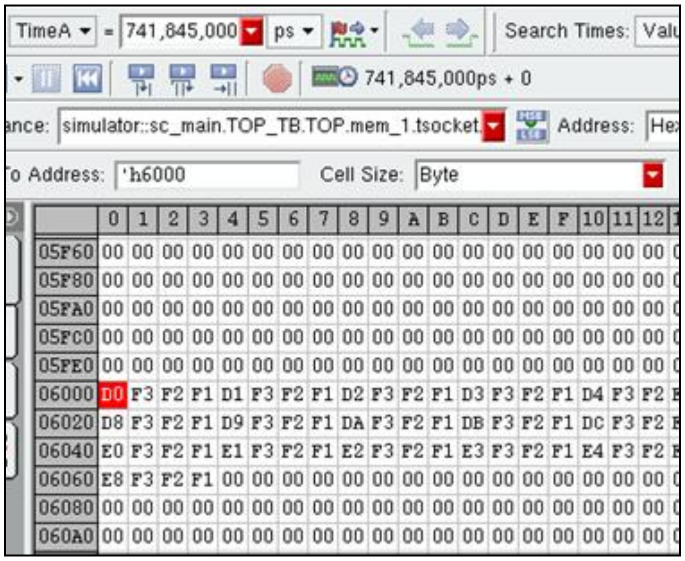
```
unsigned int n_bytes =
      socket->transport_dbg( *trans );
```

TABLE I. TLM 2.0 TRANSPORT FUNCTIONS AND MEMORY DEBUG

Function name	***descripton***	***Has side effects?***
b_transport	Blocking transport.	Yes
nb_transport_fw	Non-blocking forward transport	Yes
nb_transport_bw	Non-blocking backward transport	Yes
transport_dbg	Debug transport call	No

Since either tlm_read or tlm_write can be specified in the command field of the generic payload, the function can be used either to query the value of memory or deposit a new value. Using the command as part of a strategy to iterate over memory regions can result in a powerful data engine for memory viewing and editing tools. Figure 1 shows a sample view of memory that can be provisioned by TLM-based memory debug calls through transport_dbg.

FIGURE 1 – WINDOW DISPLAY OF TLM-BASED MEMORY VALUES



```
TimeA = 741,845,000 ps                          Search Times: Valu
741,845,000ps + 0
ance: simulator::sc_main.TOP_TB.TOP.mem_1.tsocket   Address: He
o Address: 'h6000          Cell Size: Byte

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  10 11 12
05F60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05F80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05FA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05FC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05FE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06000 D0 F3 F2 F1 D1 F3 F2 F1 D2 F3 F2 F1 D3 F3 F2 F1 D4 F3 F2
06020 D8 F3 F2 F1 D9 F3 F2 F1 DA F3 F2 F1 DB F3 F2 F1 DC F3 F2
06040 E0 F3 F2 F1 E1 F3 F2 F1 E2 F3 F2 F1 E3 F3 F2 F1 E4 F3 F2
06060 E8 F3 F2 F1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
060A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

C. *Implications of using dmi functions*

DMI functions are similar in some ways to the transport_dbg interface; however DMI is intended as a "back door" to speed up simulation time by by-passing the call chain of the transport functions for normal transactions. Although it is possible to instrument certain memory debugging operations using DMI, the TLM standard specifies the transport_dbg interface exclusively for debugging and is the preferred method for debug tools.

III. TLM2.0-BASED VIRTUAL PROTOTYPES

A. *Definitions*

There are many ways to model an SOC at the system level. For the purpose of this paper, a Virtual Prototype is a SystemC/TLM-based model that enables pre-RTL software design, verification, and system analysis before committing to hardware design. Components of a VP generally include a model of the processors, SystemC/TLM implementations of the hardware peripherals, and software such as an embedded operating system, device drivers, and application software. Device drivers and embedded software that run on the system without the operating system are called "bare metal" programs. Debugging bare metal programs is called bare metal debugging – if the debugger also deals natively with OS constructs such as threads and signals then it is said to be OS-aware.

B. *A TLM 2.0-based Virtual Prototype that runs Android*

The examples of TLM 2.0-enabled memory debug all operate on a VP of a SOC that boots Android. The example uses two ARM processor models and a TLM simple memory (RAM). Devices connected to the design have been written as SystemC models. Included in the collection of peripherals are the interrupt controller, timer, tty, audio and battery (this is just a sampling of the actual devices in the design). Figure 2 shows the system memory map for some of the devices. The memory map is an essential part of the design – it provides a mapping from logical device names to the TLM target instances including the local start and end addresses and the system base address for the device in memory.

FIGURE2. System Memory Map

device name	***TLM Target***	***Local start address***	***Local end address***	***System base address***
ram	ram.tsocket	0	0x5FFFFFF	0
Interrupt controller	interrupt.tsocket	0	0xFFF	FF00000
timer	timer.tsocket	0	0xFFF	FF01000
tty	tty0.tsocket	0	0xFFF	FF02000
audio	audio.tsocket	0	0xFFF	FF03000
battery	battery.tsocket	0	0xFFF	FF04000

There are many options for modeling the processors for this VP including using vendor or third party provided SystemC or C-based models[5] or QEMU[6]-based models. As this is a TLM-based VP, the processors are TLM initiators in the design and are modeled with processor models generated by ARM tools[7]. Once the processor models are chosen and the SystemC peripherals are ready, the development of software can begin – in particular writing drivers and porting the embedded OS. Here we've used the Android Goldfish[8] kernel. Once the OS is ported, running the system involves launching the emulator (Figure 3), booting the OS, and launching application software.

Each step in this process will typically result in modeling or software bugs, most of which are very difficult to track down without a dedicated hardware and software development environment that supports SystemC/TLM debug, bare metal debug, and OS-aware analysis and debug. The following sections show examples of three such problems and how TLM-based memory debug can help.

IV. DEBUG PROBLEMS AND SOLUTIONS THAT USE MEMORY DEBUG

A. *Finding an endianness mismatch*

The example Android VP is a collection of SystemC peripherals that communicate with ARM processor models. When working with any SOC design, it is important that the processors, the peripherals, and the embedded software agree on the endianness of data items passed between components. Endianness can be hidden at higher levels of abstraction by application software or the OS, but at the hardware and driver level, a common agreement on endianness is essential. In a VP it can be hard to keep this straight,[9] and in the low visibility environment of bare metal debug, tracking down problems caused by mismatched endianness can be very challenging.

There are at least two different ways to configure the endianness of ARM processors. This contributed to a problem encountered when developing this VP. While generating the A9 processor model, the host-endianness of the Linux development system (little-endian) was explicitly specified. This decision was made for the debug stage of the design in keeping with the TLM standard which specifies that, for optimal performance (LT models need to be fast), both initiator and target operate with little-endianness if that is the host endianness[1]. When the hardware was initially booted, the system immediately crashed. This was so early in the boot process that no traditional embedded software debugging even of assembly instructions was possible, so an investigation of the memory was undertaken to look for clues.

FIGURE 3. ANDROID

.

FIGURE 4. MEMORY VALUE – "BIG-ENDIAN"

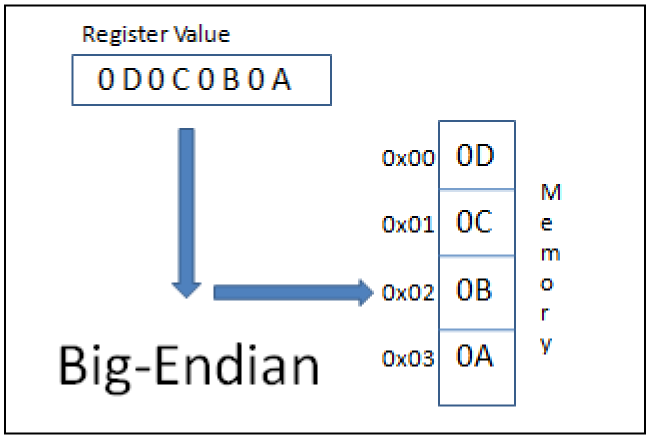
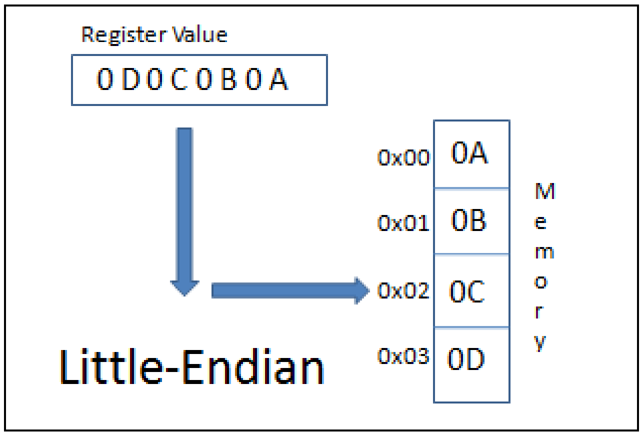


Figure 4 shows the value of a CPU register and memory obtained through TLM-based memory debug where the TLM write deposited the value of the register in memory.

FIGURE 5. MEMORY VALUE – "LITTLE-ENDIAN"

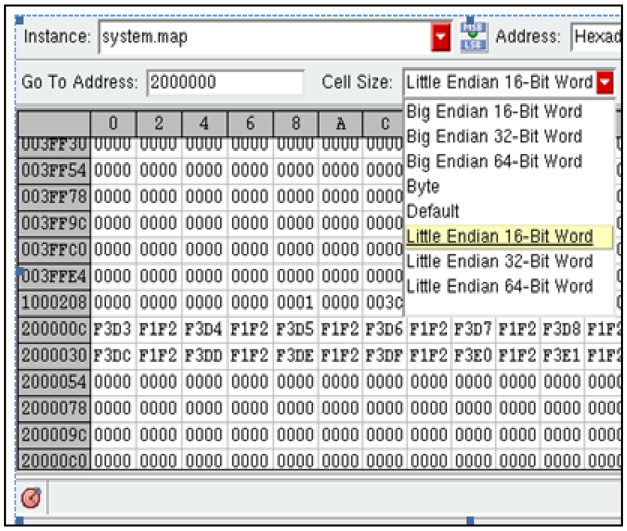


When running the bare metal design the memory content register value showed big endian representations of data (Figure 5 shows how the same register would be represented in little-endian) which caused problems when the low level drivers interpreted them as little-endian. Since the processor was configured for little endian, that was not the cause of the problem. This led to an investigation of other ways to specify the endianness of the initiator, and the ARM documentation pointed to assembly language code that when loaded into the core set it to big endian (through co-processor 15 instructions), thus overriding the setting at build time.:

```
MRC p15, 0, r0, c1, c0, 0

ORR r0, r0, #0xf8

MCR p15, 0, r0, c1, c0, 0
```

In fact this code was left over from an earlier port and should have been cleaned up as part of development. Modifying the assembly snippet fixed the problem. In general, memory debug can be used to check for endianness issues at later stages of debug as well. The user can step through the assembly code, monitoring specific memory locations looking for changes. Memory Views provisioned by TLM-based methods for gathering memory values can be configured to display memory values in different endianness, which can speed up the identification of such problems (Figure 6). Watching how the memory cells get updated can provide clues about the endianness of the data items being passed between components in the generic payload.

FIGURE 6. ENDIAN DISPLAY IN TLM MEMORY VIEW

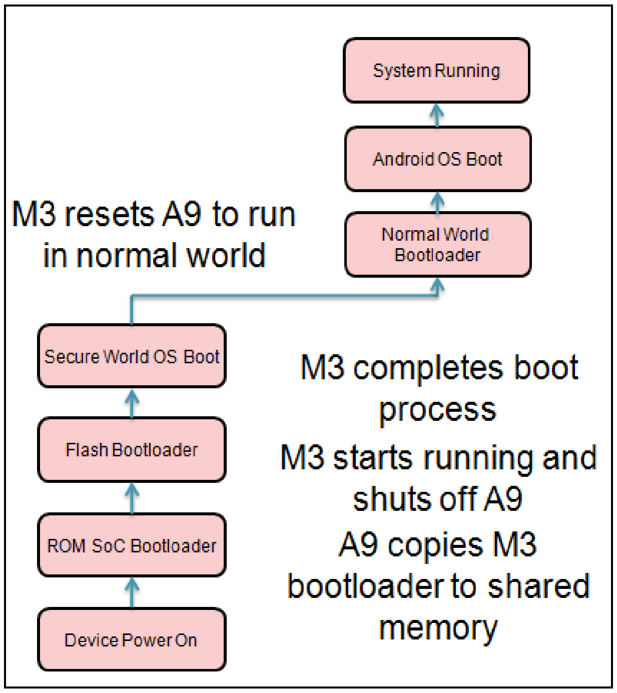


```
Instance: system.map                         Address: Hexad
Go To Address: 2000000     Cell Size: Little Endian 16-Bit Word
                                          Big Endian 16-Bit Word
        0    2    4    6    8    A    C   Big Endian 32-Bit Word
003FF30 0000 0000 0000 0000 0000 0000 0000 Big Endian 64-Bit Word
003FF54 0000 0000 0000 0000 0000 0000 0000 Byte
003FF78 0000 0000 0000 0000 0000 0000 0000 Default
003FF9c 0000 0000 0000 0000 0000 0000 0000 Little Endian 16-Bit Word
003FFC0 0000 0000 0000 0000 0000 0000 0000 Little Endian 32-Bit Word
003FFE4 0000 0000 0000 0000 0000 0000 0000 Little Endian 64-Bit Word
1000208 0000 0000 0000 0000 0001 0000 003c
200000c F3D3 F1F2 F3D4 F1F2 F3D5 F1F2 F3D6 F1F2 F3D7 F1F2 F3D8 F1F2
2000030 F3DC F1F2 F3DD F1F2 F3DE F1F2 F3DF F1F2 F3E0 F1F2 F3E1 F1F2
2000054 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
2000078 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
200009c 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
20000c0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

B. *Finding problems in the system dual- boot process*

In building this system, another problem related to the multi-phase boot sequence for the dual ARM cores. ARM provides a spec about how the boot sequence works[10] – the task is to build the system design and make sure to model all of the steps that the OS needs for the multi-phase boot sequence (Figure 7).

FIGURE 7. MULTISTAGE CORE BOOT SEQUENCE


System Running
Android OS Boot
Normal World Bootloader
M3 resets A9 to run in normal world
Secure World OS Boot
M3 completes boot process
Flash Bootloader
M3 starts running and shuts off A9
ROM SoC Bootloader
A9 copies M3 bootloader to shared memory
Device Power On

As part of the communication between the A9 and the M3, there is a special hardware reset that synchronizes communication between the cores. First the M3 boots, then it releases the A9 from reset, then the A9 boots and copies the rest of the boot-up code into memory that can be accessed by the M3. In particular, this copying requires synchronization between the two cores.

If there are problems in the boot-up phase, the system just freezes and the debug task is to isolate the problem in the visibility-deprived state of the early boot. Fortunately, access to an accurate display of memory contents can make things much easier. When this problem was encountered, examining the memory showed that the boot strap copy phase had begun, but only part of the boot-up code was successfully transferred. There could be several possible sources for this error including:

- A problem in the synchronization of accesses to the RAM being used by the A9 and M3.
- A problem with the multi-processor communication hardware block.
- A problem with the interrupt configuration.

Without automated TLM-based memory debug, the only way to investigate whether the boot-up code is successfully copied is to manually instrument the RAM block to print its values, and instrument some sort of API to specify when and how much memory to print. This is independent of "who" does this inspection request – the stimulus here is software running on the processors. TLM-based memory debug allows an on-demand request for memory contents from a component such as the RAM from a requester outside of the model code itself.

As in the endianness problem, it also helps to understand how the registers map into memory. Figure 8 shows a register view with processor registers and register values. Using the values in the address registers to identify the location of the boot-up code in RAM, and then leveraging the TLM-based memory debug to inspect the data led to the discovery that only part of the boot-up code was present – thus the code transfer failed.

With this information finding the exact cause still required several more steps. By tracing of the interrupt signals it was determined that only one interrupt was received when multiple interrupts were expected. This led to the discovery of an error in the GIC (ARM Generic Interrupt Controller) programming. The GIC code is part of the bare metal software of the design; in this case INTS[0] corresponds to GIC Interrupt ID 32, but the boot-up software was writing to a different bit in the register. After fixing the bit reference and using TLM-based memory debug to verify that the boot-up code was copied correctly, the problem was successfully fixed.

C. *Debug by redirecting Android kernel messages to memory*

While porting the Android Goldfish kernel to the VP, several port-related problems were encountered; some very early in the OS boot sequence. Android (which is Linux) provides the printk API call for kernel messages (this is part of the Android kernel logging layer which should not be confused with the user level logging modules such as liblog, logcat, etc.).

During the OS boot, many printk() messages are produced. printk() messages can be used to instrument the OS code, serve as mile posts, and to look for errors even when source or instruction debugging is not an option. In general, different implementations of Linux take different approaches to displaying printk() output, although redirection to the terminal or a file is common. Very early in the boot sequence before the TTY is initialized, terminal output is not available - the messages sit in the ring buffer in memory - and there is no straight-forward way to display the messages.

Since they exist in memory, TLM-based memory debug is a natural way to see the messages even before all of the device drivers are initialized. In Android, the ring buffer is a static array of characters:

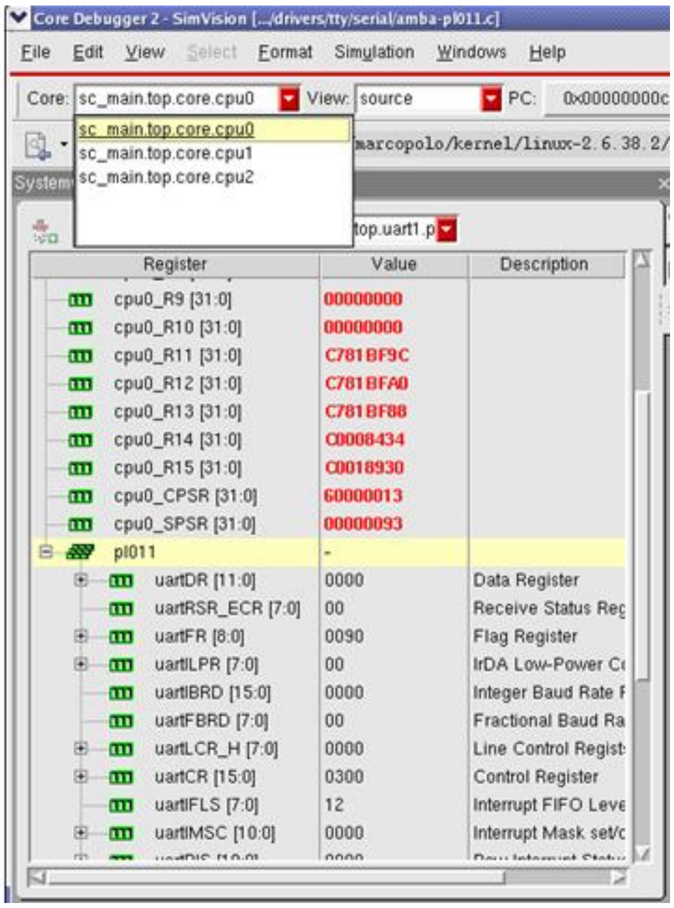
```
static char __log_buf[__LOG_BUF_LEN];

static char *log_buf = __log_buf;
```

By finding the system address of the global variable __log_buf and using TLM-based memory debug to view the contents in ASCII format, early-phase printk() messages from the kernel could be used to trace progress during the port.

V. Conclusion

Debugging a SystemC-based virtual prototype can be a difficult challenge. Problems can be caused by logic errors in the programming of the hardware, software, or both. Hardware-aware debug tools for both hardware and software debug provide a breakthrough in productivity – instead of staring at a blank xterm or a bricked phone (emulator), SOC teams now have the option of leveraging successful debug methodologies that can save days or weeks of effort. The TLM standard provides a powerful methodology for creating such tools, and in the case of memory debugging, the transport_dbg interface provides a solid core upon which to build full-featured memory debug and analysis tools for VP debug environments and IDEs.

FIGURE 8 – PROCESSOR REGISTERS AND MEMORY


Core Debugger 2 - SimVision [.../drivers/tty/serial/amba-pl011.c]
File Edit View Select Format Simulation Windows Help
System
top.uart1.p

Core: sc_main.top.core.cpu0 View: source PC: 0x00000000c

sc_main.top.core.cpu0
sc_main.top.core.cpu1
sc_main.top.core.cpu2

marcopolo/kernel/linux-2.6.38.2/

Register	Value	Description
cpu0_R9 [31:0]	00000000	
cpu0_R10 [31:0]	00000000	
cpu0_R11 [31:0]	C781BF9C	
cpu0_R12 [31:0]	C781BFA0	
cpu0_R13 [31:0]	C781BF88	
cpu0_R14 [31:0]	C0008434	
cpu0_R15 [31:0]	C0018930	
cpu0_CPSR [31:0]	60000013	
cpu0_SPSR [31:0]	00000093	
pl011	-	
uartDR [11:0]	0000	Data Register
uartRSR_ECR [7:0]	00	Receive Status Reg
uartFR [8:0]	0090	Flag Register
uartILPR [7:0]	00	IrDA Low-Power C
uartIBRD [15:0]	0000	Integer Baud Rate F
uartFBRD [7:0]	00	Fractional Baud Ra
uartLCR_H [7:0]	0000	Line Control Regist
uartCR [15:0]	0300	Control Register
uartIFLS [7:0]	12	Interrupt FIFO Leve
uartIMSC [10:0]	0000	Interrupt Mask set/c

APPENDIX A – TRANSPORT_DBG EXAMPLE

This is a simple example of a transport_dbg() implementation for a TLM 2.0-wrapped device in a virtual prototype; in this case a simple bus. First the data is validated with a simple dummy check. Then we make sure byte enable is not expected because we don't support it. Our bus can only transfer quantities of a word, so we make this check as well. We get the offset and the data pointer (dptr), then check whether the request is a read or a write. Reads correspond to memory inspection, writes correspond to value deposit. We make the call on the example_bus object to read or write the actual values in the switch. Finally we assign the data pointer and check whether there was an error during the operation. If so we set the response status to `TLM_GENERIC_ERROR_RESPONSE`.

```
unsigned int
example_bus_base_module::transport_dbg(tlm
_generic_payload& gp)
{
  tlm_response_status rspstatus;

  // Perform basic dummy test
  // on the generic_payload data.

  rspstatus = validate_request(gp, true);

  if (rspstatus != TLM_OK_RESPONSE) {
     gp.set_response_status(rspstatus);
     return (0);
  }

  // Does not support byte enable.
  if (gp.get_byte_enable_ptr()!= NULL) {
   gp.set_response_status
       (TLM_BYTE_ENABLE_ERROR_RESPONSE);
   return (0);
  }

  unsigned int dlen = p.get_data_length();
  unsigned int wordsize =
               sizeof(BUS_DATA_TYPE_32);

  // Transfer must be full word.
  if (dlen != wordsize) {
   gp.set_response_status
        (TLM_GENERIC_ERROR_RESPONSE);
        return (0);
  }

  // Get offset address
  BUS_ADDR_TYPE_64 addr =
    static_cast<BUS_ADDR_TYPE_64>
    (gp.get_address());

  BUS_ADDR_TYPE_64 offset =
example_bus_example_bus_REG_ADDR_MASK
& addr;

  BUS_DATA_TYPE_32* dptr;
  dptr =
reinterpret_cast<BUS_DATA_TYPE_32*>
    (gp.get_data_ptr());

  bool status = true;

  switch (gp.get_command()) {
   case TLM_READ_COMMAND: {
    status =this->example_bus.bus_read_dbg
           (offset, *dptr);
   break;
 }
 case TLM_WRITE_COMMAND: {
   status =this->example_bus.bus_write_dbg
         (offset, *dptr);
 break;
 }
 default :
 // Do some sort of reasonable
 // validation on this case
 break;
 }

 BUS_DATA_TYPE_32 data = *dptr;
 ::std::string cmd =
    (gp.get_command() == TLM_READ_COMMAND)
                 ? "Read" : "Write";
 if (status) {
  gp.set_response_status
              (TLM_OK_RESPONSE);
 return(dlen);
 }
 else {
  gp.set_response_status
          (TLM_GENERIC_ERROR_RESPONSE);
  return (0);
 }
}
```

References

[1] Open SystemC Initiative. "TLM-2.0 Standard." http://www.systemc.org/downloads/standards/tlm20. Retrieved December 1, 2011.

[2] Frazier, G., Motel, V., Bhatnagar, N., Larue, W. "Automatic Quantitative Analysis of Simulations of TLM 2.0 Loosely Timed Models." Proceedings of DesignCon. Feb. 2010.

[3] Frazier, G., Motel, V., Bhatnagar, N., Larue, W. "An Automatic Visual System Performance Stress Test for TLM Designs" Proceedings of DVCON Feb. 2011.

[4] Poursepanj, A. "A Common System Memory Model for SoC Software and Architecture Models using a SystemC/TLM-2.0 Interface" Proceedings of DVCON Feb. 2011.

[5] Open SystemC Initiative. "OVP Technology enabled in OSCI TLM2.0" http://www.ovpworld.org/technology_TLM2.0.php, Retrieved December 1, 2011.

[6] Cadence Design Systems. "Cadence Virtual System Platform", http://www.cadence.com/products/sd/virtual_system/pages/default.aspx. Retrieved December 1, 2011.

[7] arm.com. "Cortex-A9 Processor", http://www.arm.com/products/processors/cortex-a/cortex-a9.php. Retrieved December 1, 2011.

[8] eLinux.org. "Android on OMAP." http://elinux.org/Android_on_OMAP. Retrieved December 1, 2011.

[9] Andrews, J.. Co-*Verification of Hardware and Software for ARM SoC Design*. pgs 98-99. Elsevier. 2005.

[10] arm.com. "Arm Generic Interrupt Controller Architecture Specification", http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048a/CHDGCFJI.html Retrieved December 1, 2011.