Application Report
SPRAC12–March 2015

Debugging Tools and Techniques With IPC3.x

Angela Stegmaier

ABSTRACT

There are several useful tools and techniques that enable you to debug issues encountered when using software that leverages IPC3.x. With some guidance on the tools and techniques, you can confidently debug the IPC and the remote core software.

This application report guides the customers on the tools and techniques for debugging with IPC3.x

Contents

1 Introduction 1
2 Debug Tools 2
3 Traces 2
4 Remote Core Status Information 4
5 Using Code Composer Studio 6
6 Debugging MMU Faults and Exceptions 9
7 Other Common Issues 17
8 References 17

List of Tables

1 IPC_DEBUG Trace Levels 2
2 Remoteproc Names for DRA7XX 3
3 Remoteproc Debugfs Entries 5
4 IOMMU Entry Names 5
5 IOMMU Debugfs Entries 5

1 Introduction

During development of software, it is common to encounter issues that must be debugged to provide a robust software offering. When developing software that uses the IPC3.x product for inter-processor communication, there are tools and techniques available to aid in the debugging process. These tools and techniques help to more quickly understand and debug the issue. This document addresses all three HLOS's supported by IPC3.x: Android™ platform, Linux®, and QNX®. Where applicable, differences between the OS's are noted.

This document aims to provide tools, techniques, and resources for debugging issues encountered when using the IPC3.x to communicate with remote core software.

NOTE: This document assumes that the reader is familiar with the IPC3.x product and its interfaces. For documentation and further details, see the IPC3.x release. Releases can be found at the IPC 3.x download page. For a link to the download page for IPC 3.x releases, see [1].

2 Debug Tools

There are several tools that can be leveraged for debugging the IPC, from simple tracing to using Code Composer Studio™ (CSS) (see [2]) to attach to the remote cores. Each provides different advantages in the debugging process.

2.1 Tracing

When an issue occurs, checking the traces is often the first thing done. The traces can give quick insight into what may be happening. In normal, non-error cases, you might see little to no traces from the IPC. But when there is an error, there often is an error trace that can be used to shed light on the issue. An error trace often gives an error code and a module or function name that can be used to identify where the error was thrown.

2.2 Linux and Android - Debugfs

Another useful place to look for debug info is in the debugfs. Information for each remote processor can be found there, as well as remote core traces, state information, and more.

2.3 Code Composer Studio

Connecting over JTAG using Code Composer Studio provides the ability to see exactly what is happening on the remote core, providing access to a wealth of information including viewing memory, registers, and stepping through the code.

3 Traces

The first place to check when an issue occurs is the traces. Often an error code or a trace regarding an error gives some clue. Both the remote core traces and the HLOS-side traces can be checked.

3.1 Linux and Android - Enabling IPC Traces

By default, the IPC only prints error traces. To enable additional tracing, use the IPC_DEBUG environment variable at runtime. This feature is only supported on Linux and is available starting in IPC 3.22.00.05. Table 1 lists the two levels of tracing supported.

Table 1. IPC_DEBUG Trace Levels

Trace Level	Description
1	Enables all warnings and errors to be printed
2	Turns on all tracing (including socket and LAD client tracing). Warning: this can be very "chatty."

Refer to [3].

3.2 QNX – Enabling IPC Traces

When using QNX, additional traces are enabled by setting environment variables. There are separate environment variables for enabling traces in the resource manager and the user libraries.

3.2.1 Resource Manager Traces

Enable additional traces in the slog by setting the environment variable IPC_DEBUG_SLOG_LEVEL at runtime. By default, only errors and warnings are printed. The IPC_DEBUG_SLOG_LEVEL can be set before launching IPC to enable more traces. Setting the level to 7 enables all IPC traces. The default level is 2.

```
# export IPC_DEBUG_SLOG_LEVEL=7
```

These traces are printed to the slog, and can be viewed by using the sloginfo command. All IPC traces use 42 as identification in the slog, and you can filter the slog to view only these traces:

```
# sloginfo –m42
```

3.2.2 User Library Traces

User library traces can be enabled by using the variable IPC_DEBUG when launching the application. Valid levels are 1 to 3, with 3 being the most verbose. For example:

```
# IPC_DEBUG=<level> app_host
```

3.3 Linux and Android - Remote Core Traces

The remote core traces can be checked by using debugfs. Run the following command, replacing the “X” with the core-id for the remote core to be checked.

```
cat /d/remoteproc/remoteprocX/trace0
```

Check the following when checking the traces after an error recovery has occurred:

```
cat /d/remoteproc/remoteprocX/trace0_last
```

This provides the last traces that happened before error recovery was triggered. For more information about debugging remote core faults and exceptions, see *Debugging MMU Faults and Exceptions*.

The core-id, “X”, starts at 0 and increments to include all of the remote cores supported by the remoteproc module in the dts file. Because the number of remoteprocs supported can vary depending on the dts configuration, it is not ensured that a certain remote core will always have a certain core-id if the remoteprocs supported in the dts file changes.

The core associated with a particular core-id can be found by checking the name of the remoteproc (see Linux and Android - Remoteproc). Table 2 associates the remoteproc name with the common name of the remote processor.

```
cat /d/remoteproc/remoteprocX/name
```

Table 2. Remoteproc Names for DRA7XX

Debugfs Name	Remote Core Name
58820000.ipu	IPU1
55020000.ipu	IPU2
40800000.dsp	DSP1
41000000.dsp	DSP2

3.4 QNX - Remote Core Traces

The remote core traces can be checked by using sysfs. Run the following command, replacing <core name> with the core name for the remote core to check. Valid core names for DRA7XX are IPU1, IPU2, DSP1, and DSP2.

```
# cat /dev/ipc-trace/<core name>
```

When checking the traces after error recovery has happened, check the logfile specified at IPC startup, if one was specified. When starting IPC, specify a logfile using the “-c” option. When an error recovery happens, the last traces are dumped to this log file. For more information about debugging remote core faults and exceptions, see *Debugging MMU Faults and Exceptions*.

3.5 Adding Traces

In some cases, you may want to add traces to get more information about the issue. If the issue is reliably reproducible, one technique is to add additional traces to get more information. If the issue is timing-related, this technique may not be helpful, as it may mask the issue.

3.5.1 SYS-BIOS

You can add traces in the SYS-BIOS IPC code that come to the trace buffer by using the System_printf() API. After adding traces to the SYS-BIOS IPC and rebuilding the IPC, you must rebuild the remote core image. The traces come to the remote core trace buffer and can be viewed by following the instructions in *Linux and Android - Remote Core Traces*.

3.5.2 Linux

Additionally, traces can be added in the Linux code. The modules of interest when adding traces are the remoteproc, iommu, and rpmsg modules in the kernel, and the MessageQ, MMRPC, and LAD modules in the user space.

In the kernel are these modules in the following paths:

- drivers/remoteproc/
- drivers/iommu/
- drivers/rpmsg/

Most of the user space code can be found in the IPC package, in the linux folder. The MMRPC code is found in the *packages/ti/ipc/mm/* folder.

4 Remote Core Status Information

Useful information about the status of the remote cores can be found in debugfs.

4.1 Linux and Android - Remoteproc

Information about each remote core can be found in the following, where the “X” can be replaced with the remote core id.

```
# cat /d/remoteproc/remoteprocX/<entry>
```

Table 3 lists what can be found for each core.

Table 3. Remoteproc Debugfs Entries

Entry	Description
name	Processor name, comprised of the RAM address and the processor type, (for example, 58820000.ipu for IPU1). For a complete list of names, see Table 2.
recovery	Returns either “enabled” or “disabled”, indicating if recovery is enabled or disabled for the remote processor.
state	Gives the state of the remote processor. State is one of: • offline (0) • suspended (1) • running (2) • crashed (3)
trace0	Returns the contents of the remote processor trace buffer.
trace0_last	Created after recovering the remote core. Returns the contents of the remote processor trace buffer before recovery was triggered.
version	Returns the version. Currently returns nothing.

4.2 Linux and Android - IOMMU Info

Information about the IOMMU can also be found in debugfs. It can be found in the following path, where “XXXXXXXX” is replaced by the register address for the remote core MMU registers.

```
# cat /d/omap_iommu/XXXXXXXX.mmu/<entry>
```

Table 4 gives the corresponding core name for each MMU for DRA7XX, and the register address in the TRM.

Table 4. IOMMU Entry Names

IOMMU Entry	Core Name
58882000.mmu	IPU1
55082000.mmu	IPU2
40d01000.mmu	DSP1 (MMU1)
40d02000.mmu	DSP1 (MMU2)
41501000.mmu	DSP2 (MMU1)
41502000.mmu	DSP2 (MMU2)

Some of this information is inaccessible from a suspended state.

Table 5 lists what can be found for each core.

Table 5. IOMMU Debugfs Entries

Entry	Description
nr_tlb_entries	Gives the number of tlb entries
pagetable	Dumps the pagetable entries
regs	Gives the values of the MMU registers
tlb	Lists the tlb entries.

4.3 QNX – Remote Core State Information

Find out the current state of the remote core by issuing the following command:

```
# cat /dev/ipc-state/<core_name>
```

The “core_name” is the name of the remote core. Valid names for DRA7XX are IPU1, IPU2, DSP1, and DSP2. The current state will show as “running” or “reset”.

5 Using Code Composer Studio

A useful tool for debugging issues is Code Composer Studio. CCS allows easy connection to the remote core in order to see the state of the remote core.

5.1 Debug Symbols

The remote core image must be built with debug symbols to see information such as the call stack and variables. Once attached, load the symbols. The symbols are built into the executable itself. When loading symbols, point to the same executable that is loaded on the target (or the unstripped version locally, if it is stripped to save space when loading to the target).

5.2 Linux and Android - Disabling Remoteproc Auto-Suspend

You may want to disable auto-suspend of the remote cores (provided that is not what is being debugged). When the core is suspended, you will not be able to connect to the remote core using CCS. Auto-suspend can be disabled by setting the power control to “on” for the remote core.

```
#echo on > /sys/bus/platform/devices/<device>/power/control
```

The remote core device name for each remote core can be found in Table 2.

5.3 Linux and Android – Disabling Watchdog

You may decide to disable the watchdog timers when debugging and using CCS. Otherwise, while connected to the target, the watchdog may expire, triggering an abort sequence. Disable the watchdog timers for a remote core by removing their definitions from the dts file. For example, to disable the watchdog timers for IPU1, change the dts file as below:

```
&ipu1 {
                status = "okay";
                memory-region = <&ipu1_cma_pool>;
     mboxes = <&mailbox5 &mbox_ipu1_legacy>;
     timers = <&timer11>;
-               watchdog-timers = <&timer7>, <&timer8>;
+               /*watchdog-timers = <&timer7>, <&timer8>;*/
```

5.4 SYS/BIOS – Disabling Watchdog

When using QNX, you can disable the watchdog from within the remote core image itself.

If using Linux or Android, this step is not required; simply follow the instructions in *Linux and Android - Disabling Watchdog*.

To disable the usage of the watchdog from the remote core without completely disabling the device exception module (DEH), comment out the calls to Watchdog_init in the SYS/BIOS IPC code. These calls can be found in packages/ti/deh/Deh.c, packages/ti/deh/DehDsp.c, and packages/ti/ipc/ipcmgr/IpcMgr.c.

packages/ti/deh/Deh.c:

```
/*
 *  ======== Deh_Module_startup ========
 */
Int Deh_Module_startup(Int phase)
{
    if (AMMU_Module_startupDone() == TRUE) {
-        Watchdog_init(ti_sysbios_family_arm_m3_Hwi_excHandlerAsm__I);
+        //Watchdog_init(ti_sysbios_family_arm_m3_Hwi_excHandlerAsm__I);
        return Startup_DONE;
    }

    return Startup_NOTDONE;
}
```

packages/ti/deh/DehDsp.c:

```
Int Deh_Module_startup(Int phase)
{
#if defined(HAS_AMMU)
    if (AMMU_Module_startupDone() == TRUE) {
-        Watchdog_init((Void (*)(Void))ti_sysbios_family_c64p_Exception_handler);
+        //Watchdog_init((Void (*)(Void))ti_sysbios_family_c64p_Exception_handler);
        return Startup_DONE;
    }

    return Startup_NOTDONE;
#else
-    Watchdog_init((Void (*)(Void))ti_sysbios_family_c64p_Exception_handler);
+    //Watchdog_init((Void (*)(Void))ti_sysbios_family_c64p_Exception_handler);

    return Startup_DONE;
#endif
```

packages/ti/ipc/ipcmgr/IpcMgr.c:

```
Void IpcMgr_rpmsgStartup(Void)
{
    Assert_isTrue(MultiProc_self() != MultiProc_getId("HOST"), NULL);
    RPMessage_init(MultiProc_getId("HOST"));

-#ifdef IpcMgr_USEDEH
+#if 0
    /*
     * When using DEH, initialize the Watchdog timers if not already done
     * (i.e. late-attach)
     */
#ifdef IpcMgr_DSP
    Watchdog_init((Void (*)(Void))ti_sysbios_family_c64p_Exception_handler);
#elif IpcMgr_IPU
    Watchdog_init(ti_sysbios_family_arm_m3_Hwi_excHandlerAsm__I);
#endif
#endif
}

[...]

Void IpcMgr_ipcStartup(Void)
{
```

```
    UInt procId = MultiProc_getId("HOST");
    Int status;

    /* TransportRpmsgSetup will busy wait until host kicks ready to recv: */
    status = TransportRpmsgSetup_attach(procId, 0);
    Assert_isTrue(status >= 0, NULL);

    /* Sets up to communicate with host's NameServer: */
    status = NameServerRemoteRpmsg_attach(procId, 0);
    Assert_isTrue(status >= 0, NULL);

-#ifdef IpcMgr_USEDEH
+#if 0
    /*
     * When using DEH, initialize the Watchdog timers if not already done
     * (i.e. late-attach)
     */
#ifdef IpcMgr_DSP
    Watchdog_init((Void (*)(Void))ti_sysbios_family_c64p_Exception_handler);
#elif IpcMgr_IPU
    Watchdog_init(ti_sysbios_family_arm_m3_Hwi_excHandlerAsm__I);
#endif
#endif
}
```

Following this, rebuild the IPC and the remote core image to have an image with DEH, but without watchdog enabled.

5.5 Attaching Before the Issue

In certain cases, you may want to attach to the remote core before the issue has occurred. If the issue is reliably reproducible and always occurs at the same location, then adding a breakpoint close to where the issue happens could be a good way to get a better picture of what is happening.

One instance where it may be difficult is if the issue is happening during boot-up of the remote core. In this case, it may be necessary to add a while loop in the main function, to attach before the issue occurs. Add a loop similar to this:

```
{
    volatile int foo = 1;
    while(foo);
}
```

Then, after attaching, load the symbols, add the breakpoints, change “foo” to 0, and continue running.

5.6 Attaching After the Issue

You can also attach to the core after the issue, load the symbols, and see the state and view memory. You can view the Exception module’s exception CallStack ROV view and the task module’s per task CallStack ROV view. For more information about the runtime object viewer (ROV) in the RTSC documentation online, see [9].

5.7 Viewing the State of the Remote Core

Once attached and with symbols loaded, the state of the processor can be inspected. You can see the program counter, memory windows, registers, call stack, and the ROV, among other things. For more information about the runtime object viewer (ROV), check the RTSC documentation online (see [9]).

6 Debugging MMU Faults and Exceptions

Errors commonly manifest as MMU faults, exceptions, and watchdog errors (if using a version of IPC with watchdog available and enabled).

6.1 Linux and Android - Disabling Error Recovery

To debug an error, it may be necessary to turn of error recovery. Error recovery can be disabled by giving the following command:

```
echo disabled > /d/remoteproc/remoteprocX/recovery
```

See *Linux and Android - Remoteproc* for more information.

6.2 QNX – Disabling Error Recovery

To disable error recovery on QNX using IPC version 3.22 and above, give the -d option when launching the ipc binary. For example:

```
    # ipc –d IPU2 dra7x-ipu2-fw.xem4
```

6.3 Crash Dump

If any of these three errors are encountered, you will get a crash dump from the remote core which is visible in the remote core traces. If error recovery is disabled, the dump can be found in trace0 (when using Linux/Android) or in /dev/ipc-trace/<core_name> (when using QNX); otherwise, the trace is found in trace0_last (when using Linux/Android) and in the logfile (when using QNX).

An example of the crash dump will look like this:

```
[0][     91.045] Exception occurred at (PC) = 0000c976
[0][     91.045] CPU context: thread
[0][     91.045] BIOS Task name: {empty-instance-name} handle: 0x80060090.
[0][     91.045] BIOS Task stack base: 0x800600e0.
[0][     91.045] BIOS Task stack size: 0x800.
[0][     91.045] [t=0x18f6df13] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1078: E_hardFault:
FORCED
[0][     91.045] ti.sysbios.family.arm.m3.Hwi: line 1078: E_hardFault: FORCED
[0][     91.045] [t=0x18f9a0cb] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1155: E_busFault:
PRECISERR: Immediate Bus Fault, exact addr known, address: 96000000
[0][     91.045] ti.sysbios.family.arm.m3.Hwi: line 1155: E_busFault: PRECISERR: Immediate Bus
Fault, exact addr known, address: 96000000
[0][     91.045] R0 = 0x96000000  R8  = 0xffffffff
[0][     91.045] R1 = 0x00000000  R9  = 0xffffffff
[0][     91.045] R2 = 0x00000000  R10 = 0xffffffff
[0][     91.045] R3 = 0x80060814  R11 = 0xffffffff
[0][     91.045] R4 = 0x00013098  R12 = 0x8006074c
[0][     91.045] R5 = 0x0000000a  SP(R13) = 0x80060820
[0][     91.045] R6 = 0xffffffff  LR(R14) = 0x0000c973
[0][     91.045] R7 = 0xffffffff  PC(R15) = 0x0000c976
[0][     91.045] PSR = 0x61000000
[0][     91.045] ICSR = 0x00438803
[0][     91.045] MMFSR = 0x00
[0][     91.045] BFSR = 0x82
[0][     91.045] UFSR = 0x0000
[0][     91.045] HFSR = 0x40000000
[0][     91.045] DFSR = 0x00000000
[0][     91.045] MMAR = 0x96000000
[0][     91.045] BFAR = 0x96000000
[0][     91.045] AFSR = 0x00000000
```

```
[0][     91.045] Stack trace
[0][     91.045]  00 [op faaaf00e] 00006abd (ret from call to 00015010)
[0][     91.045]  01 [op ff49f005] 00006ac3 (ret from call to 0000c954)
[0][     91.045]  -- [op 98009000] 000154c9
[0][     91.045]  -- [op 00000000] 000a0001
[0][     91.045]  -- [op 80084a64] 0000fec9
[0][     91.045]  -- [op 0001a75c] 000068b9
[0][     91.045]  -- [op 80084a64] 0000fec9
[0][     91.045]  -- [op bd0ef919] 00015b91
[0][     91.045] Stack dump base 800600e0 size 2048 sp 80060820:
[0][     91.045] 80060820: 00000001 00006abd 96000000 00000000  ffffffff 00006ac3 0000000a
00006bf4
[0][     91.045] 80060840: 00000000 00000000 80041800 80060ab0  00000080 56414c53 50495f45
be003155
[0][     91.045] 80060860: bebebebe bebebebe bebebebe bebebebe  bebebebe bebebebe bebebebe
bebebebe
[0][     91.045] 80060880: bebebebe bebebebe bebebebe bebebebe  bebebebe 00000000 00000000
00000001
[0][     91.045] 800608a0: 00000001 000154c9 0001309a 0000000a  00000000 80041820 00000001
ffffffff
[0][     91.045] 800608c0: ffffffff 0000fec9 00000000 00000000  000068b9 0000fec9 00015b91
bebebebe
[0][     91.045] Terminating execution...
```

6.4 Exception Dump Decoding

Some useful information that can be found in the dump is the fault address, PC address, register contents, and call stack.

When an error occurs, you gets a crash dump from the remote core that looks similar to the one in *Crash Dump*.

The particular dump example above is from a MMU read fault. This dump provides important information in helping to understand what has happened. Some of the useful parts are broken down in the following section.

6.4.1 Exception Dump Breakdown

6.4.1.1 Timestamp

All traces (not just exception dumps) provide a timestamp for each trace. The time starts from the booting of the remote core. The timestamp is highlighted in Figure 1.

```
[0][     91.045] Exception occurred at (PC) = 0000c976
[0][     91.045] CPU context: thread
[0][     91.045] BIOS Task name: {empty-instance-name} handle: 0x80060090.
```

Figure 1. Trace Timestamps

The timestamp information can be useful even in non-crash situations, indicating the amount of time taken between two events. You can add traces at each event and then see when the events run.

For example, to check that a certain event is happening every second, put a trace at that event, then check the timestamps to see that it is happening as expected.

6.4.1.2 PC Address

The PC address where the exception occurred is also provided (see Figure 2). This can be used, in conjunction with the map file or CCS, to identify the line of code where the exception happened.

```
[0][     91.045] Exception occurred at (PC) = 0000c976
[0][     91.045] CPU context: thread
[0][     91.045] BIOS Task name: {empty-instance-name} handle: 0x80060090.
```

Figure 2. Crash Dump PC Address

6.4.1.3 Task Information

The information about the task that was executing when the exception occurred is also provided (see Figure 3).

```
[0][     91.045] Exception occurred at (PC) = 0000c976
[0][     91.045] CPU context: thread
[0][     91.045] BIOS Task name: {empty-instance-name} handle: 0x80060090.
[0][     91.045] BIOS Task stack base: 0x800600e0.
[0][     91.045] BIOS Task stack size: 0x800.
```

Figure 3. Crash Dump Task Information

6.4.1.4 Fault Information

The information about the fault is also provided (see Figure 4). This can look different depending on the type of exception that occurred, but often provides a fault address to identify the source of the fault.

```
[0][     91.045] Exception occurred at (PC) = 0000c976
[0][     91.045] CPU context: thread
[0][     91.045] BIOS Task name: {empty-instance-name} handle: 0x80060090.
[0][     91.045] BIOS Task stack base: 0x800600e0.
[0][     91.045] BIOS Task stack size: 0x800.
[0][     91.045] [t=0x18f6df13] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1078: E_hardFault:
FORCED
[0][     91.045] ti.sysbios.family.arm.m3.Hwi: line 1078: E_hardFault: FORCED
[0][     91.045] [t=0x18f9a0cb] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1155: E_busFault:
PRECISERR: Immediate Bus Fault, exact addr known, address: 96000000
[0][     91.045] ti.sysbios.family.arm.m3.Hwi: line 1155: E_busFault: PRECISERR: Immediate Bus
Fault, exact addr known, address: 96000000
```

Figure 4. Crash Dump Fault Information

6.4.1.5 Registers

A dump of the register contents at the time of the exception is also provided (see Figure 5).

```
[0][     91.045] [t=0x18f6df13] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1078: E_hardFault:
FORCED
[0][     91.045] ti.sysbios.family.arm.m3.Hwi: line 1078: E_hardFault: FORCED
[0][     91.045] [t=0x18f9a0cb] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1155: E_busFault:
PRECISERR: Immediate Bus Fault, exact addr known, address: 96000000
[0][     91.045] ti.sysbios.family.arm.m3.Hwi: line 1155: E_busFault: PRECISERR: Immediate Bus
Fault, exact addr known, address: 96000000
[0][     91.045] R0 = 0x96000000  R8  = 0xffffffff
[0][     91.045] R1 = 0x00000000  R9  = 0xffffffff
[0][     91.045] R2 = 0x00000000  R10 = 0xffffffff
[0][     91.045] R3 = 0x80060814  R11 = 0xffffffff
[0][     91.045] R4 = 0x00013098  R12 = 0x8006074c
[0][     91.045] R5 = 0x0000000a  SP(R13) = 0x80060820
[0][     91.045] R6 = 0xffffffff  LR(R14) = 0x0000c973
[0][     91.045] R7 = 0xffffffff  PC(R15) = 0x0000c976
[0][     91.045] PSR = 0x61000000
[0][     91.045] ICSR = 0x00438803
[0][     91.045] MMFSR = 0x00
[0][     91.045] BFSR = 0x82
[0][     91.045] UFSR = 0x0000
[0][     91.045] HFSR = 0x40000000
[0][     91.045] DFSR = 0x00000000
[0][     91.045] MMAR = 0x96000000
[0][     91.045] BFAR = 0x96000000
[0][     91.045] AFSR = 0x00000000
[0][     91.045] Stack trace
[0][     91.045]  00 [op faaaf00e] 00006abd (ret from call to 00015010)
[0][     91.045]  01 [op ff49f005] 00006ac3 (ret from call to 0000c954)
[0][     91.045]  -- [op 98009000] 000154c9
[0][     91.045]  -- [op 00000000] 000a0001
[0][     91.045]  -- [op 80084a64] 0000fec9
[0][     91.045]  -- [op 0001a75c] 000068b9
[0][     91.045]  -- [op 80084a64] 0000fec9
[0][     91.045]  -- [op bd0ef919] 00015b91
[0][     91.045] Stack dump base 800600e0 size 2048 sp 80060820:
[0][     91.045] 80060820: 00000001 00006abd 96000000 00000000  ffffffff 00006ac3 0000000a
00006bf4
[0][     91.045] 80060840: 00000000 00000000 80041800 80060ab0  00000080 56414c53 50495f45
be003155
[0][     91.045] 80060860: bebebebe bebebebe bebebebe bebebebe  bebebebe bebebebe bebebebe
bebebebe
[0][     91.045] 80060880: bebebebe bebebebe bebebebe bebebebe  bebebebe 00000000 00000000
00000001
[0][     91.045] 800608a0: 00000001 000154c9 0001309a 0000000a  00000000 80041820 00000001
ffffffff
[0][     91.045] 800608c0: ffffffff 0000fec9 00000000 00000000  000068b9 0000fec9 00015b91
bebebebe
[0][     91.045] Terminating execution...
```

Figure 5. Crash Dump Register Contents

6.4.1.6 Stack Trace

The stack trace is also provided (see Figure 6). This can be used in conjunction with the source code and the map file or CCS to get more information about what was executing at the time of the crash.

```
 [0][     91.045] R0 = 0x96000000  R8  = 0xffffffff
[0][     91.045] R1 = 0x00000000  R9  = 0xffffffff
[0][     91.045] R2 = 0x00000000  R10 = 0xffffffff
[0][     91.045] R3 = 0x80060814  R11 = 0xffffffff
[0][     91.045] R4 = 0x00013098  R12 = 0x8006074c
[0][     91.045] R5 = 0x0000000a  SP(R13) = 0x80060820
[0][     91.045] R6 = 0xffffffff  LR(R14) = 0x0000c973
[0][     91.045] R7 = 0xffffffff  PC(R15) = 0x0000c976
[0][     91.045] PSR = 0x61000000
[0][     91.045] ICSR = 0x00438803
[0][     91.045] MMFSR = 0x00
[0][     91.045] BFSR = 0x82
[0][     91.045] UFSR = 0x0000
[0][     91.045] HFSR = 0x40000000
[0][     91.045] DFSR = 0x00000000
[0][     91.045] MMAR = 0x96000000
[0][     91.045] BFAR = 0x96000000
[0][     91.045] AFSR = 0x00000000
[0][     91.045] Stack trace
[0][     91.045]  00 [op faaaf00e] 00006abd (ret from call to 00015010)
[0][     91.045]  01 [op ff49f005] 00006ac3 (ret from call to 0000c954)
[0][     91.045]  -- [op 98009000] 000154c9
[0][     91.045]  -- [op 00000000] 000a0001
[0][     91.045]  -- [op 80084a64] 0000fec9
[0][     91.045]  -- [op 0001a75c] 000068b9
[0][     91.045]  -- [op 80084a64] 0000fec9
[0][     91.045]  -- [op bd0ef919] 00015b91
[0][     91.045] Stack dump base 800600e0 size 2048 sp 80060820:
[0][     91.045] 80060820: 00000001 00006abd 96000000 00000000  ffffffff 00006ac3 0000000a
00006bf4
[0][     91.045] 80060840: 00000000 00000000 80041800 80060ab0  00000080 56414c53 50495f45
be003155
[0][     91.045] 80060860: bebebebe bebebebe bebebebe bebebebe  bebebebe bebebebe bebebebe
bebebebe
[0][     91.045] 80060880: bebebebe bebebebe bebebebe bebebebe  bebebebe 00000000 00000000
00000001
[0][     91.045] 800608a0: 00000001 000154c9 0001309a 0000000a  00000000 80041820 00000001
ffffffff
[0][     91.045] 800608c0: ffffffff 0000fec9 00000000 00000000  000068b9 0000fec9 00015b91
bebebebe
[0][     91.045] Terminating execution...
```

Figure 6. Crash Dump Stack Trace

6.4.2 MMU Faults

MMU faults occur when an address that is not mapped to the remote core MMU is accessed. This can be due to a read, write, or an attempt to execute the address. When an MMU fault occurs, a crash dump from the remote core occurs that looks similar to the example provided in Crash Dump.

Some debugging techniques, as well as common times when an MMU fault occurs, are given as examples in the following sections.

6.4.2.1 Using CCS to Halt the Code When the Fault Happens

If the fault always happens at the same address, pre-map the location and then set up CCS with a breakpoint for that address. In this way, you can view the state of the remote core when the fault happens and see the call stack. From there, put a breakpoint at the surrounding code and step through to see where the fault happens.

Pre-mapping the address can be done either through the remote core resource table, or through CCS. With CCS, you can connect to the debug DAP and then bring up a memory window to inspect the MMU registers. Directly program the MMU from here to map some unused memory to the fault address location.

For example:

- MMU CAM: 0x9600000E (Change the most significant 20 bits here to match the fault address. For example, it would be 96000 if the fault address is 0x96000010)
- MMU RAM: 0xBA300000 (Change the most significant 20 bits here to match an unused 4-KB physical region in the memory map)
- MMU Lock: 0x00000400
- MMU LD: 0x00000001

6.4.2.2 Using the Crash Dump to Find the Location of the Fault

The crash dump call stack can indicate where the crash occurred. Using that information, connect to the remote core with CCS and put a breakpoint in the code at the most recent function in the call stack before the crash. From there, step through the code until the crash happens.

6.4.2.3 Example – Accessing a Memory Region That is not Mapped

When using the L2 MMU, every address accessed by the remote core must be mapped. An attempt to access an un-mapped address results in an MMU fault. The following example explores the crash dump of an access to an un-mapped area.

Here is an example fault dump:

```
[0][    107.092] Exception occurred at (PC) = 0000cfa6
[0][    107.092] CPU context: thread
[0][    107.092] BIOS Task name: {empty-instance-name} handle: 0x80060050.
[0][    107.092] BIOS Task stack base: 0x800600a0.
[0][    107.092] BIOS Task stack size: 0x800.
[0][    107.092] [t=0x16e623ee] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1078: E_hardFault:
FORCED
[0][    107.092] ti.sysbios.family.arm.m3.Hwi: line 1078: E_hardFault: FORCED
[0][    107.092] [t=0x16e8e328] ti.sysbios.family.arm.m3.Hwi: ERROR: line 1155: E_busFault:
PRECISERR: Immediate Bus Fault, exact addr known, address: 96000000
[0][    107.092] ti.sysbios.family.arm.m3.Hwi: line 1155: E_busFault: PRECISERR: Immediate Bus
Fault, exact addr known, address: 96000000
[0][    107.092] R0 = 0x96000000  R8  = 0xffffffff
[0][    107.092] R1 = 0x00000000  R9  = 0xffffffff
[0][    107.092] R2 = 0x00000000  R10 = 0xffffffff
[0][    107.092] R3 = 0x800607d4  R11 = 0xffffffff
[0][    107.092] R4 = 0xffffffff  R12 = 0x8006070c
[0][    107.092] R5 = 0xffffffff  SP(R13) = 0x800607e0
[0][    107.092] R6 = 0xffffffff  LR(R14) = 0x0000cfa3
[0][    107.092] R7 = 0xffffffff  PC(R15) = 0x0000cfa6
[0][    107.092] PSR = 0x61000000
[0][    107.092] ICSR = 0x00438803
[0][    107.092] MMFSR = 0x00
[0][    107.092] BFSR = 0x82
[0][    107.092] UFSR = 0x0000
[0][    107.092] HFSR = 0x40000000
[0][    107.092] DFSR = 0x00000000
[0][    107.092] MMAR = 0x96000000
[0][    107.092] BFAR = 0x96000000
[0][    107.092] AFSR = 0x00000000
[0][    107.092] Stack trace
[0][    107.092]  00 [op f8c6f00e] 00006bcd (ret from call to 00014d58)
[0][    107.092]  -- [op 0f960010] 0000ef71
[0][    107.092]  01 [op f9d9f006] 00006bd3 (ret from call to 0000cf84)
[0][    107.092]  02 [op 00015fbd] 80060864
[0][    107.092]  -- [op 00000000] 000affff
[0][    107.092]  -- [op 00000014] 00010211
[0][    107.092]  -- [op 0000c504] 000069c9
[0][    107.092]  -- [op 00000014] 00010211
[0][    107.092]  -- [op bd0efa01] 00015899
[0][    107.092] Stack dump base 800600a0 size 2048 sp 800607e0:
[0][    107.092] 800607e0: 00000001 00006bcd 96000000 00000000  0000ef71 00006bd3 0000000a
00006d04
[0][    107.092] 80060800: 00000000 00000000 80040200 80060a40  00000080 56414c53 50495f45
be003155
[0][    107.092] 80060820: bebebebe bebebebe bebebebe bebebebe  bebebebe bebebebe bebebebe
bebebebe
[0][    107.092] 80060840: bebebebe bebebebe bebebebe bebebebe  bebebebe 00000000 00000000
00000001
[0][    107.092] 80060860: 00000001 000198c5 ffffffff 0000000a  00000000 80040220 00000001
ffffffff
[0][    107.092] 80060880: ffffffff 00010211 00000000 00000000  000069c9 00010211 00015899
bebebebe
[0][    107.092] Terminating execution...
```

Figure 7. Crash Dump Stack Trace

From the crash dump, the fault address is 0x96000000. The address will not be found in the resource table, which is why the fault occurred.

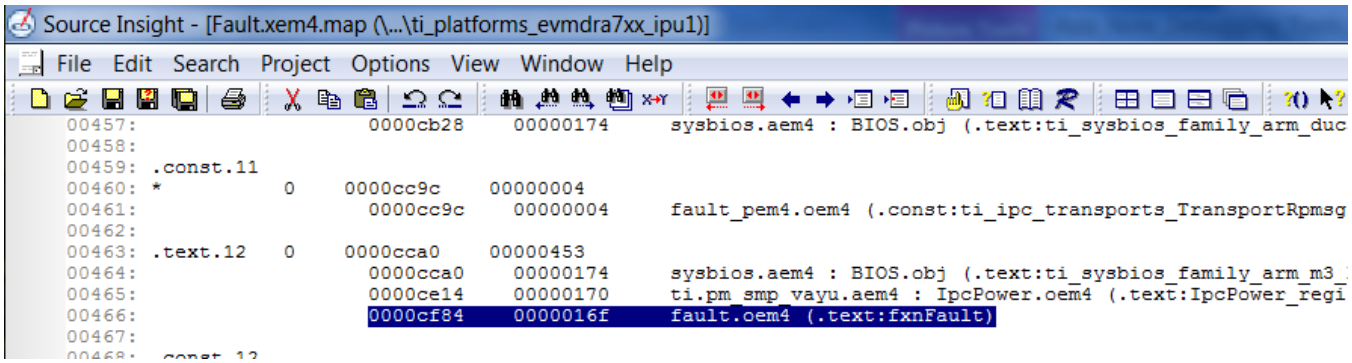
Avoid hard-coding of virtual addresses for peripherals and memory blocks with a one-time physical to virtual address lookup using the resource table. There is an API available for this called Resource_physToVirt() in the resource module. This alerts that the address is not mapped in the resource table when the translation fails.

From here, either use the crash dump to see the PC and call stack or follow the instructions in *Using CCS to Halt the Code When the Fault Happens*. Error recovery, watchdog timers, and remoteproc autosuspend may need to be disabled to connect CCS. See *Linux and Android - Disabling Error Recovery*, *Linux and Android - Disabling Watchdog*, and *Linux and Android - Disabling Remoteproc Auto-Suspend* for more information on disabling these.

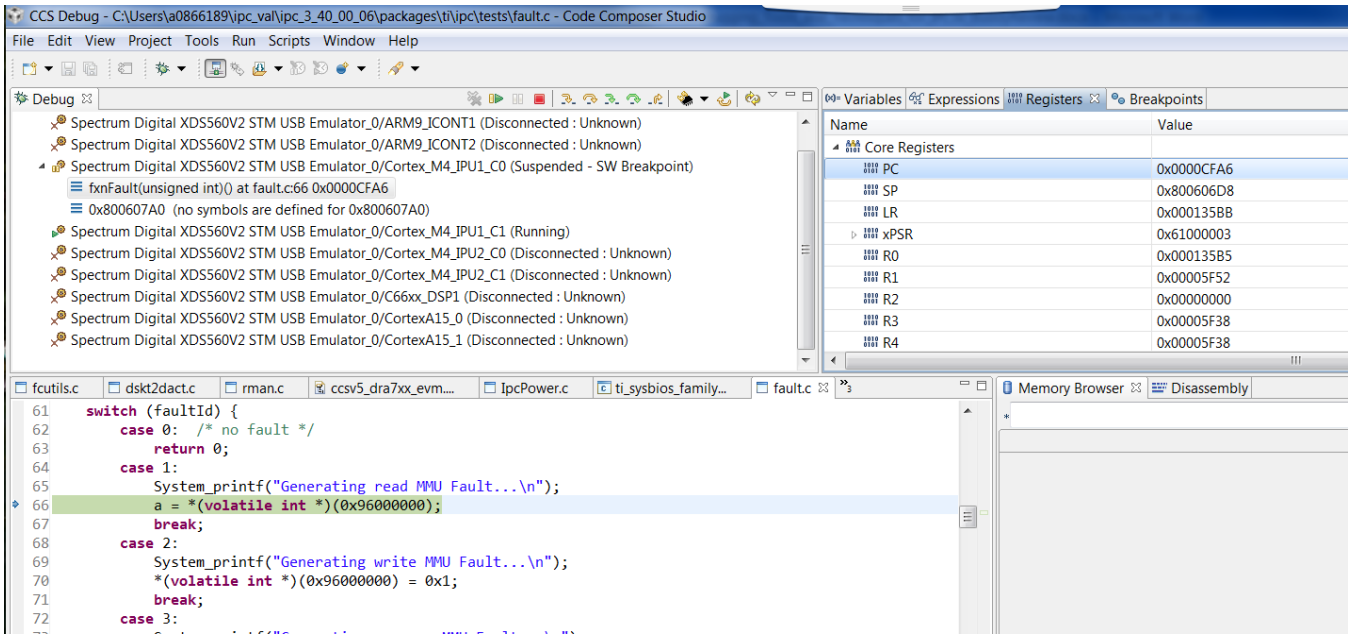
For this example, use the PC address which, as seen in the crash dump, is at 0xcfa6.

```
[0][    107.092] Exception occurred at (PC) = 0000cfa6
```

Find the corresponding function by looking this address up in the map file for the remote core image. If the PC address is invalid due to an issue such as stack corruption, then this may not yield useful results. In this case, something useful is found:



Alternatively, use CCS to see the location of the fault. If CCS was already connected to the remote core before the fault happened, the core will have halted in the abort function. From here, directly set the PC address and see the line that caused the fault:



Use this technique at any time after booting the remote core to see what a PC address corresponds to. It will display the line that caused the error. This may, however, prevent proper execution because the registers and call stack won't have proper values.

You can now isolate the particular line in the *fxnFault()* function that was executing. That code is found in the file <ipc_package>/packages/ti/ipc/tests/fault.c:

```
        case 1:
            System_printf("Generating read MMU Fault...\n");
            a = *(volatile int *)(0x96000000);
            break;
```

Figure 8. Fault Generating Code

Figure 8 clearly shows what caused the fault in this code, but when it is not clear, use CCS to see the fault in action. Once connected and the symbols are loaded, put a breakpoint in this function (if this function does not happen often). Once the breakpoint is hit, step through the code to find what is causing the fault.

Upon stepping through, observe that the variable, a, is being set to the contents of 0x96000000, which is equal to the fault address. This is the fault in this example.

Next, decide if this is a valid value that needs mapping, or if this is an invalid value that passed due to some error in the code. If it is still not known where the value is coming from, use CCS to trace it back through the call stack to fix the code. If it turns out that the address is an address that must be accessible to the remote core, then map it through the resource table to the appropriate physical memory.

7 Other Common Issues

7.1 IPC Versions Across Processors

Often, issues arise due to version mismatches between the IPC versions running on different cores. These issues can manifest in various ways, so ensure that the version of the IPC running on the host is the same version running on the remote core that the host is communicating with. All versions must match, especially when updating one image or the other.

7.2 Android and Linux - Late Attach and IPC

When using the late-attach feature of the IPC, the u-boot and kernel must both be configured with late-attach enabled or both configured without late-attach enabled. Having a mismatch can lead to crashes and other undesired behavior.

For example, if the kernel is configured to have late attach enabled, but u-boot has not loaded the remote processor, then the kernel will crash when the kernel tries to access a register it assumes was already configured by u-boot.

More information on late attach for Android can be found at http://processors.wiki.ti.com/index.php/Early_Boot_and_Late_Attach.

More information on late attach for Linux can be found in the GLSDK Software Developer's Guide: http://downloads.ti.com/infotainment/esd/jacinto6/glsdk/latest/exports/DRA7xx_GLSDK_Software_Developers_Guide.html#Using_the_Late_attach_functionality.

8 References

Much of the information contained in this application report can be found in the following wiki links and download pages. These wikis are kept up-to-date and can be referenced for further information.

1. IPC Product Releases – http://downloads.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ipc/
2. Download CCS – http://processors.wiki.ti.com/index.php/Download_CCS
3. IPC DEBUG – http://processors.wiki.ti.com/index.php/IPC_DEBUG
4. Tracing – http://processors.wiki.ti.com/index.php/IPC_Install_Guide_QNX#Tracing
5. IPC MMU fault debug – http://processors.wiki.ti.com/index.php/IPC_MMU_fault_debug
6. IPC Slave Error Recovery – http://processors.wiki.ti.com/index.php/IPC_Slave_Error_Recovery
7. Exception Dump Decoding Using the CCS Register View – http://processors.wiki.ti.com/index.php/SYS/BIOS_FAQs#4_Exception_Dump_Decoding_Using_the_CCS_Register_View

8. Debugging Crashes – http://processors.wiki.ti.com/index.php/Early_Boot_and_Late_Attach#Debugging_Crashes
9. Runtime Object Viewer – http://rtsc.eclipse.org/docs-tip/Runtime_Object_Viewer

Additionally, see a list of all the IPC-related wiki pages by searching the IPC category at http://processors.wiki.ti.com/index.php/Category:IPC.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265