Qualcomm Technologies, Inc.

Qualcomm® FastRPC

User Guide

80-N7039-2 B

March 31, 2017

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	December 2016	Initial release
B	March 2017	Numerous changes to support the SDM660 chipset.

Contents

1 Introduction

1.1 Purpose

This document describes how to use the Qualcomm® FastRPC framework and debug certain issues.

1.2 Conventions

Code variables appear in angle brackets, for example, <number>.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 FastRPC overview

A Remote Procedure Call (RPC) allows a computer program calling a procedure to execute in another remote processor, while hiding the details of the remote interaction.

FastRPC is the Qualcomm-proprietary RPC mechanism used to enable remote function calls between the CPU and aDSP. Hexagon Access customers using Hexagon Vector eXtension (HVX) algorithms can use the FastRPC framework because the HVX functions are called in the CPU, while the actual execution is on the aDSP.

2.1 FastRPC framework

Use the FastRPC framework for all programs involving customer-written modules that are to be called by the CPU but are to be executed on the aDSP.

Specifically, all HVX use cases in camera streaming, and computer vision applications employ the FastRPC framework. Therefore, use FastRPC when working on camera streaming or computer vision use cases that involve HVX algorithms to be executed on the aDSP.

2.2 FastRPC architecture

The FastRPC framework exists in the User process domain in both the CPU and aDSP.

NOTE: The following graphic has been updated.

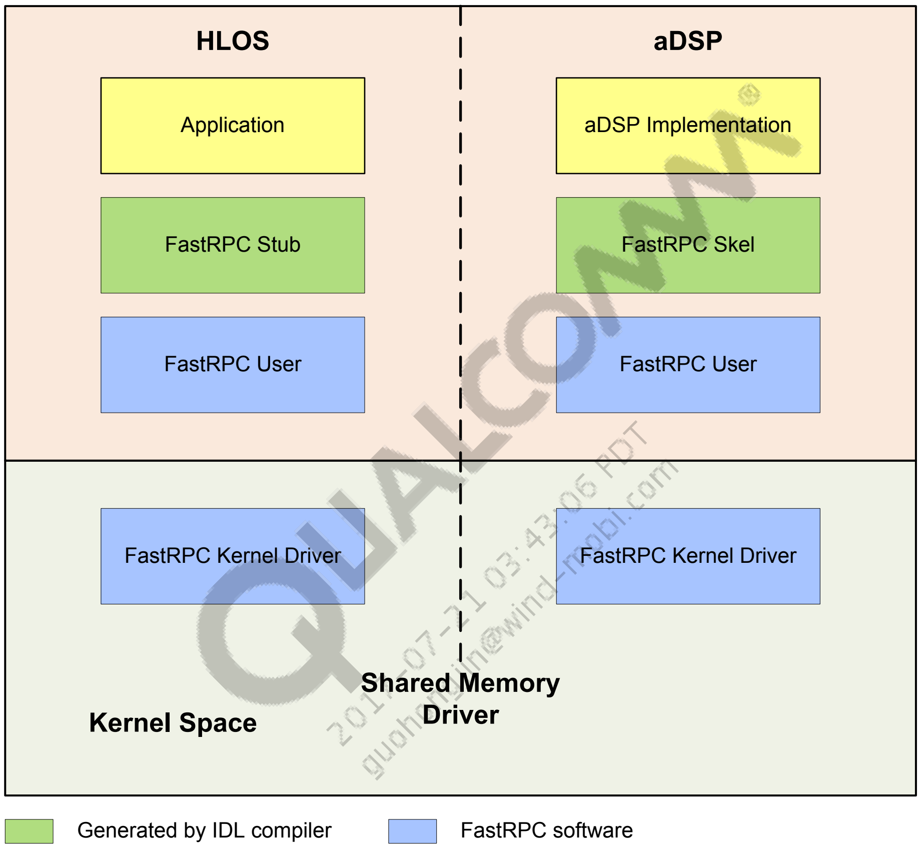


Figure 2-1 FastRPC framework among other software components

For information on the other components in the diagram, refer to the documents listed in Section A.1.

2.2.1 DSP protection domain and DSP user protection domain

Because the aDSP is a real-time processor whose stability critically affects the overall user experience, different protection domains (PDs) exist in the aDSP software architecture. These PDs ensure the stability of the kernel software and the safety of Qualcomm proprietary hardware information.

There are three protection domains in the aDSP.

- Kernel – Access to all memory of all PDs. The memory footprint of this PD must be the smallest footprint of all PDs.
- Guest OS – Access to the memory of its own PD, the memory of the User PD, and some system registers. Many Qualcomm drivers use this PD.
- User – Access only to the memory of its own PD.

The aDSP system firmware automatically makes system calls to the Guest OS or Kernel PD, if necessary. Customer FastRPC programs run in the User PD.

2.2.2 Android ION allocator

The ION allocator is a contiguous memory allocator provided by the Android platform. It can allocate a contiguous memory region that both the CPU and aDSP can share.

Use the ION allocator to configure the memory size, alignment, heap ID where memory will be allocated, and special configuration flags.

For information on memory management, see Section 2.4. For more information on the ION allocator, refer to the Hexagon SDK[1] document page:

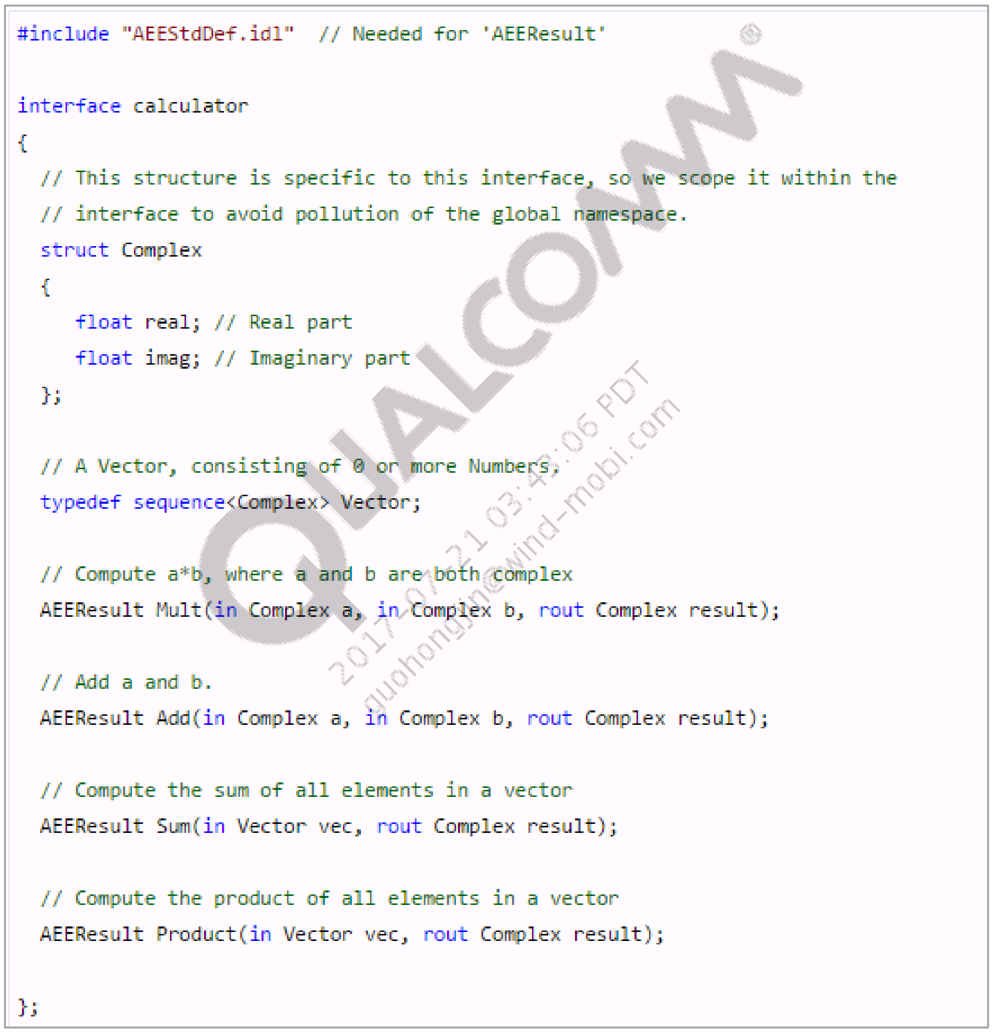
```
<HEXAGON_SDK_ROOT>/docs/Technologies_FastRPC.html#Using%20the%20ION%20al
locator
```

[1] In this document, Hexagon SDK refers to versions 3.0 and later, unless indicated otherwise.

2.2.3 IDL compiler

Interfaces for the aDSP platform and all FastRPC programs are described in a language called IDL. IDL allows interface authors to expose only what that object does, but not where it resides or the programming language in which it is implemented. IDL provides flexibility of software implementation while maintaining a consistent interface for the software module.

Following is a typical IDL header file.



```
#include "AEEStdDef.idl"  // Needed for 'AEEResult'

interface calculator
{
  // This structure is specific to this interface, so we scope it within the
  // interface to avoid pollution of the global namespace.
  struct Complex
  {
     float real; // Real part
     float imag; // Imaginary part
  };

  // A Vector, consisting of 0 or more Numbers.
  typedef sequence<Complex> Vector;

  // Compute a*b, where a and b are both complex
  AEEResult Mult(in Complex a, in Complex b, rout Complex result);

  // Add a and b.
  AEEResult Add(in Complex a, in Complex b, rout Complex result);

  // Compute the sum of all elements in a vector
  AEEResult Sum(in Vector vec, rout Complex result);

  // Compute the product of all elements in a vector
  AEEResult Product(in Vector vec, rout Complex result);

};
```

When using the function parameters:

- Indicate input parameters as **in**.
- Indicated parameters to be modified as output as **rout**.

For more information on the concept and use of the IDL compiler, refer to the Hexagon SDK document page:

```
<HEXAGON_SDK_ROOT>/docs/Tools_IDL%20Compiler.html
```

2.3 How FastRPC works

The FastRPC framework is a typical proxy pattern. The interface object stub and the implementation skeleton objects are on different processors. FastRPC clients are directly exposed to the stub object, and the skeleton object is called internally to the FastRPC framework.

The FastRPC framework consists of the following components.

Component	Description
Client	User mode process that initiates the remote invocation, typically the customer process.
Stub	Autogenerated code linked with the User mode process that marshals parameters
ADSPRPC driver	aDSP RPC kernel driver that receives the remote message invocations, queues the messages, and then waits for the response after signaling the remote side.
ADSPRPC framework	aDSP RPC framework dequeues the messages from the queue and dispatches them for processing.
Skel	Autogenerated code that unmarshaling parameters
Object	Method implementation.

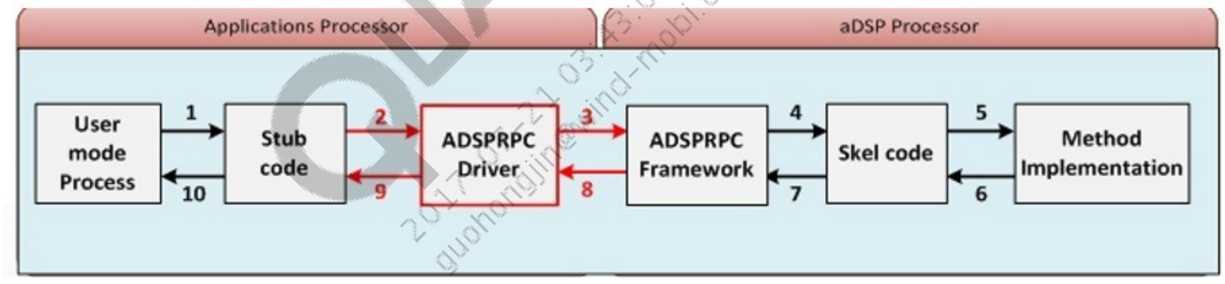


Figure 2-2 Interaction of the FastRPC components

1. The User mode process calls the stub version of the function.

   The stub code converts the function call to an RPC message.
2. The stub code internally invokes the ADSPRPC driver on the applications processor to queue the converted message.
3. The ADSPRPC driver on the applications processor sends the queued message to the ADSPRPC framework on the aDSP.
4. The ADSPRPC framework on the aDSP dispatches the relevant skeleton code.
5. The skeleton code unmarshals the parameter and calls the method implementation.
6. The skeleton code waits for implementation to finish processing, and, in turn, marshals the return value into the return message.
7. The skeleton code calls the ADSPRPC framework to queue the return message to be transmitted to the applications processor.

8. The ADSPRPC framework on the aDSP sends the return message back to the ADSPRPC driver on the applications processor.
9. ADSPRPC driver identifies the waiting stub code and dispatches the return value.
10. The stub code unmarshals the return message and sends it to the calling User mode process.

2.4 Memory buffer management

The aDSP is part of the SoC package. Therefore, various processor units (in this case, the aDSP and CPU) have access to the same hardware memory unit (such as DDR3).

For better memory control, there are multiple logical divisions of the memory. Each processing unit has exclusive and shared access to various memory areas. Memory protection units (MPUs) implement the access control.

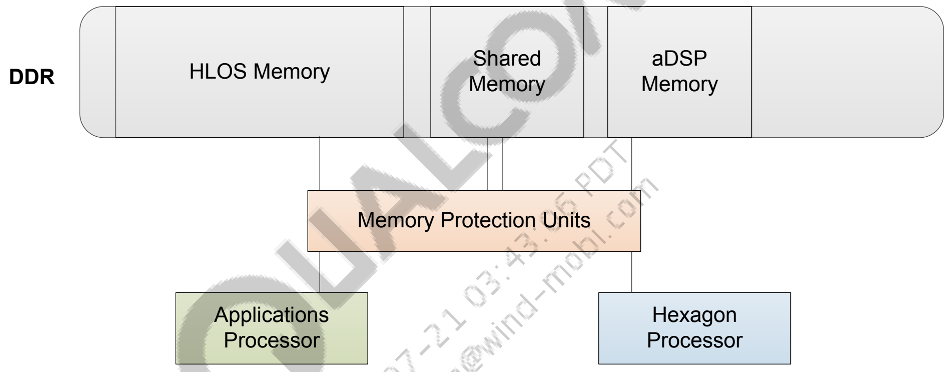


Figure 2-3 Relationship of MPUs and memory buffers

The shared memory is used separately in the CPU and aDSP via different memory translation. The same region or regions of physical memory are translated into different virtual addresses in each processing unit via different translation lookahead buffers (TLBs).

As with all standard computers, there is a limited number of TLB entries. If the entries are all filled, each new memory mapping causes eviction of an existing TLB entry. Adding and evicting TLB entries eventually slows the entire program, especially when it is avoidable. Therefore, we strongly recommend allocating large physically contiguous chunks of buffers to be mapped to the corresponding large virtual address range. Do not map several physically non-contiguous chunks to represent a virtual contiguous range.

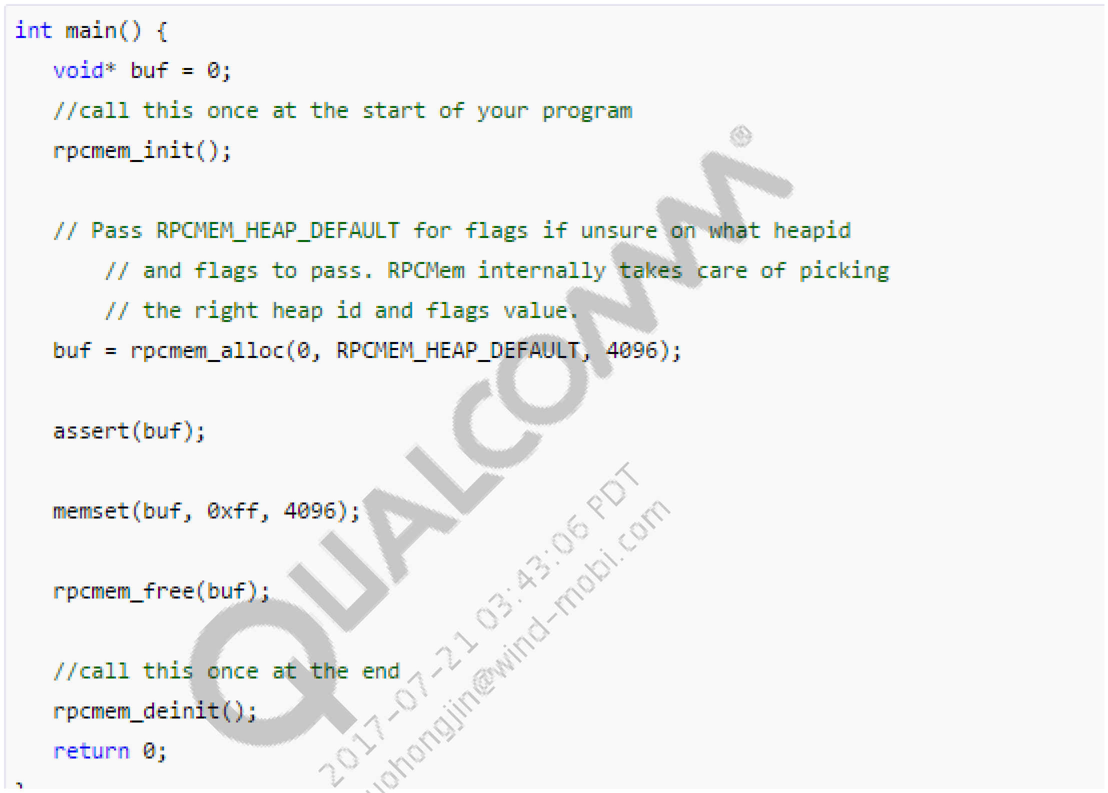
In the latest high-end Qualcomm products, a System Memory Management Unit (SMMU) introduces another translation layer:

- It optimizes the need to allocate small chunks of memory when larger chunks are not available
- It optimizes the need to allocate contiguous physical addresses to minimize the number of TLB entries

With the SMMU layer, the actual non-contiguous memory chunks are presented in a contiguous view to each processing unit. However, the SMMU records and gathers the scattered data in its own records.

NOTE: Nothing is required from the software side to configure SMMU functionality.

To ensure contiguous physical memory, use the ION allocator (Section 2.2.2). Following is an example of rpcmem usage.



```
int main() {
   void* buf = 0;
   //call this once at the start of your program
   rpcmem_init();

   // Pass RPCMEM_HEAP_DEFAULT for flags if unsure on what heapid
       // and flags to pass. RPCMem internally takes care of picking
       // the right heap id and flags value.
   buf = rpcmem_alloc(0, RPCMEM_HEAP_DEFAULT, 4096);

   assert(buf);

   memset(buf, 0xff, 4096);

   rpcmem_free(buf);

   //call this once at the end
   rpcmem_deinit();
   return 0;
```

For more information on memory management in the FastRPC framework, refer to the Hexagon SDK document:

```
<HEXAGON_SDK_ROOT>/docs/Technologies_FastRPC.html#RPCMem
```

2.5 Cache operation

In all multiprocessor computer systems, cache coherency (or cache synchronization) means the local cache of the shared resources must be up-to-date when accessed, thus ensuring the correct functionality. The FastRPC framework provides the following methods to ensure cache coherency, which is related to the IDL compiler (Section 2.2.3):

- If a parameter is designated as ***in*** in the IDL builder, the CPU flushes the cache for the buffer corresponding to the parameter. Then it makes an RPC call, where the DSP invalidates the cache for the buffer before reading it.
- If a parameter is designated as ***rout*** in the IDL builder, the CPU makes an RPC call. The DSP flushes the cache after writing to the buffer that corresponds to the parameter. Then the CPU invalidates the cache for the buffer after the RPC call returns and before reading the buffer.
- If a parameter is designated as ***inrout*** in the IDL builder, the cache operations for both *in* and *rout* are executed.

Threads and processes

Both the CPU and aDSP employ multithreaded operating systems, so it is important to understand the thread and process operations related in each FastRPC call.

- A separate process is created on the DSP for each process on high-level operating system (HLOS), which is the OS running on the CPU.
  - Each process or thread on the HLOS has a corresponding process or thread on the DSP.
  - This process is created when the device is opened on the HLOS, and it is destroyed when the device is closed on the HLOS.
- The shell process called **fastrpc_shell_0** is loaded on the DSP when a user process is spawned. The object of this shell must be compiled together with the DSP build.
- When an RPC message is invoked and no corresponding thread exists on the DSP, the required thread is created on the DSP.
- The threads are destroyed when the corresponding HLOS thread exits.

More details are provided with debugging examples in Chapter 4.

2.6 Call flows

Marking a parameter as *in* incurs a different call flow from *rout*.

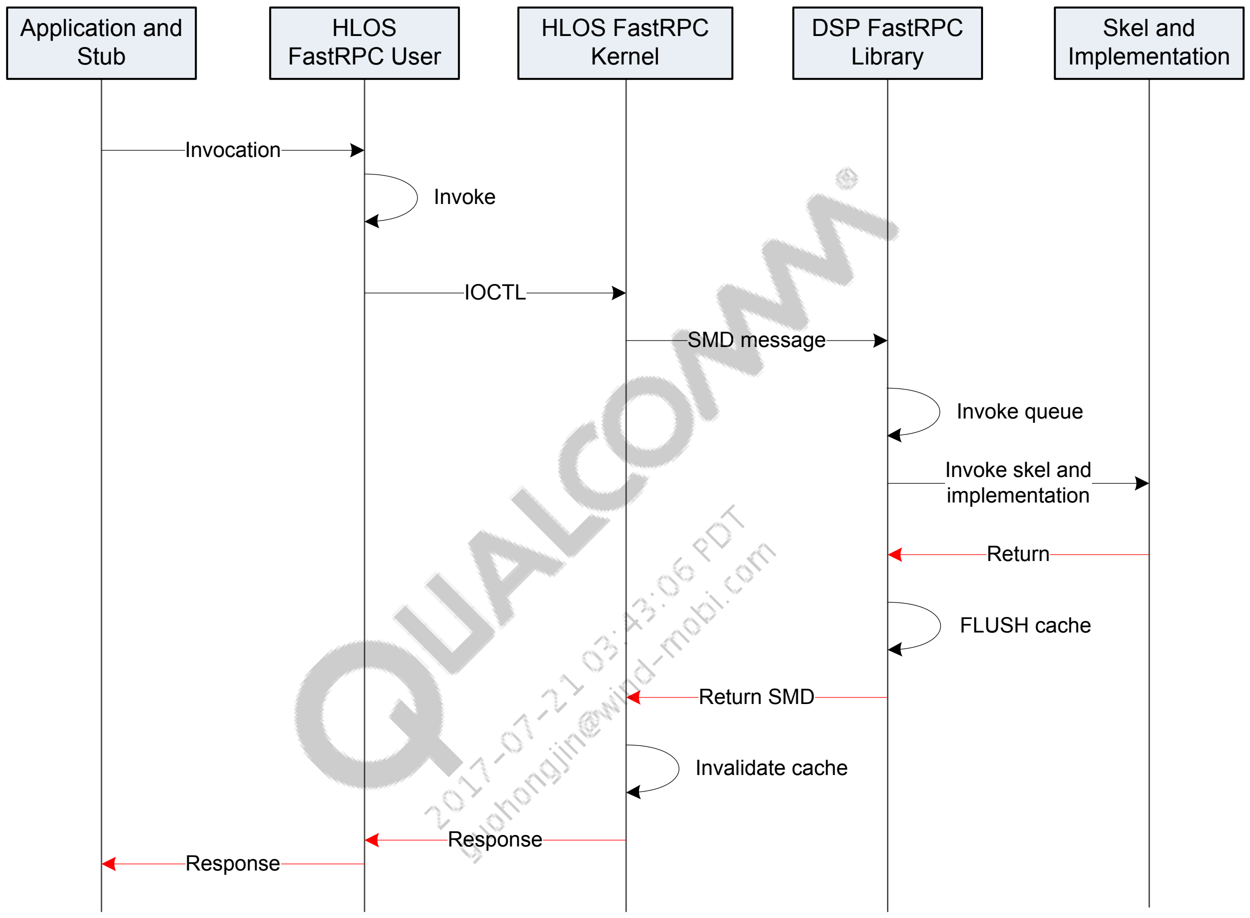


Figure 2-4 Call flow: a parameter is designated as *in*

NOTE: In the call flow diagrams, *IOCTL* is the same as the IOCTL in all Linux kernel drivers.

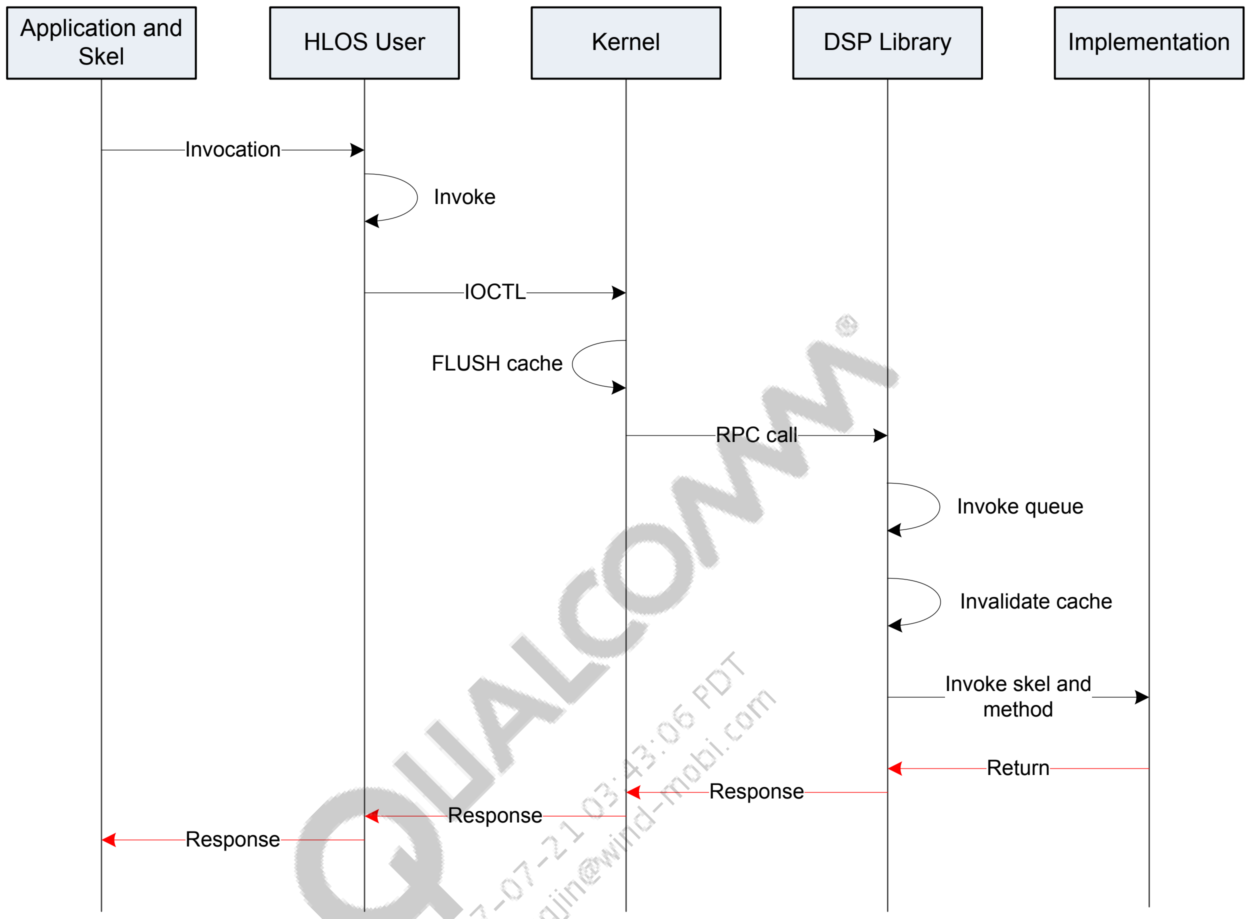


Figure 2-5 Call flow: a parameter is designated as *rout*

NOTE: A parameter designated as *inrout* will invoke cache coherency operations applied to both *in* and *rout*. Thus, for *inrout*, add extra cache operations in both the *in* and *rout* call flows.

2.7 IO coherency

NOTE: This section was added to this document revision.

For each FastRPC invocation (whether the call type is *in*, *rout*, or *inrout*), both the CPU and aDSP must flush and invalidate the cache to maintain cache coherency in the system. Results might be additional latency in the FastRPC call and additional overhead for the entire system performance.

IO coherency eliminates the cache flush and cache invalidations. In Figure 2-5, for example, the flush operation is implemented as fast forwarding (snooping) from the DSP cache to the CPU cache. This implementation replaces the requirement for the cache to be flushed and then reloaded from the DDR, thus avoiding the otherwise necessary memory read/write operations. Similar steps are executed for bus invalidation.

IO coherency has the following advantages:

- It reduces the time spent on cache cleaning or invalidation.
- It is a hardware feature, allowing the aDSP to snoop into the CPU cache.

Section 4.2.3.1 has a table that lists IO coherency enabled for the MSM8998 chipset. This table shows a significant improvement in FastRPC latency compared to other targets where IO coherency is not enabled.

2.8 Code organization

NOTE: This section was added to this document revision.

- HLOS
- Kernel driver – `kernel/drivers/char/adsprpc.c`
- Built as part of the LA kernel image
- User space – `vendor/qcom/proprietary/adsprpc`
- Shared object library: libadsprpc.so
- Daemon process: adsprpcd
- aDSP
- `adsp_proc/platform/*`
- ADSPRPC framework library that is linked with the aDSP image

  The library acts as the transport that accepts remove invocations originating from applications processor.

3 FastRPC operations

The calculator example illustrates how to use FastRPC. Only those areas directly related to FastRPC are highlighted. For more information about the calculator example, refer to the Hexagon SDK document:

```
<SDK root>\docs\calculator_android.html
```

3.1 Use rpcmem

In the calculator example, `calculator_test.c` is compiled on the CPU, and `calculator_imp.c` is compiled on the aDSP. They are the stub-skeleton pair necessary to form the FastRPC components in the CPU and aDSP.

Following is the recommended usage of rpcmem.

1. In `calculator_test.c::45`:

   ```
   rpcmem_init();
   ```

   - For convenience, when trying FastRPC examples, we recommend using rpcmem functions instead of the ION functions that are provided in the Android source code.
   - The HLOS code must take care of rpcmem operations, not the DSP code.
   - Initialize rpcmem before doing anything.

2. In `calculator_test.c::49`:

   ```
   printf("- allocate %d bytes from ION heap\n", len);
     if (0 == (test = (int*)rpcmem_alloc(0, RPCMEM_HEAP_DEFAULT, len))) {
       printf("Error: alloc failed\n");
       nErr = 1;
       goto bail;
     }
   ```

3. In `test_calculator.c::95`:

   ```
   if (test) {
       rpcmem_free(test);
     }
     rpcmem_deinit();
   ```

4. In `android.min:13` and `android.min:22`, the following code links the RPC library:

   ```
   calculator_test_DLLS += libcalculator libadsprpc
   ```

3.2 Call FastRPC functions

If the function is defined in `calculator_imp.c`., calling a FastRPC function does not require anything special.

For example, in `calculator_imp.c::10`:

```
int calculator_sum(const int* vec, int vecLen, int64* res)
```

In `calculator_test.c::69`:

```
if (0 != calculator_sum(test, num, &result)) {
      printf("Error: compute on aDSP failed\n");
      nErr = 1;
      goto bail;
}
```

3.3 Push user shared objects

The calculator_walkthrough.py script provides a general understanding of what is required to compile and run the calculator examples. The following locations show the built executables and objects on the CPU and aDSP.

1. In `calculator_walkthrough.py::58-61`:

   Push the Android executable for any standalone applications into `/data/`. Push the Android libraries, or the stub shared objects, into `/system/lib`.

   ```
   os.system('adb push '+calculator_exe+' /data')
   os.system('adb shell chmod 777 /data/calculator')
   os.system('adb push '+libcalculator+' /system/lib')
   ```

   Where in `libcalculator_walkthrough.py::29-30`:

   ```
   calculator_exe=HEXAGON_SDK_ROOT+'/examples/common/calculator/android_Deb
   ug/ship/calculator'
   libcalculator=HEXAGON_SDK_ROOT+'/examples/common/calculator/android_Debu
   g/ship/libcalculator.so'
   ```

2. In `calculator_walkthrough.py::65`:

   Push the DSP libraries, or the skeleton shared objects, into `/system/lib/rfsa/adsp`:

   ```
   os.system('adb push '+libcalculator_skel+' /system/lib/rfsa/adsp')
   ```

   Where in `calculator_walkthrough.py::31`:

   ```
   libcalculator_skel=HEXAGON_SDK_ROOT+'/examples/common/calculator/hexagon
   _Debug_dynamic/ship/libcalculator_skel.so'
   ```

3.4 Sign shared objects

To enhance the security of the shared objects, and to protect the data security of the entire software system, digitally sign all shared objects that are built.

In `calculator_walkthrough.py::34`:

```
android_Debug = ‘make –C’ + HEXAGON_SDK_ROOT +
‘/examples/common/calculator tree V=android_Debug || exit /b’
```

This example illustrates how to build the calculator Android objects.

1. Comment out `calculator_walkthrough.py::49`:

   ```
   os.system(call_test_sig),
   ```

2. In `calculator_walkthrough.py::46`:

   ```
   call_test_sig=’python ‘+ HEXAGON_SDK_ROOT+’/scripts/testsig.py’
   ```

Given steps 1 and 2, the calculator example will run normally.

However, by changing `calculator_walkthrough.py::34` as follows:

```
android_Debug = ‘make –C’ + HEXAGON_SDK_ROOT +
‘/examples/common/calculator tree V=android_Release || exit /b’
```

In this example, the calculator example will not run unless `calculator_walkthrough.py::46` is uncommented.

These examples show the importance of signing the user-generated shared objects. Chapter 4 provides more details on how to detect problems where a shared object is not signed.

3.5 Mark buffers as uncached

To minimize latency, mark the ION buffers to be allocated as uncached if the following condition is true:

Other than DSP HVX processing, the HLOS does not access the allocated buffer (the HLOS neither reads from nor writes to the allocated buffer).

Continuing with the calculator example, use the following function and parameters to allocate the RPC memory buffer:

```
rpcmem_alloc(0, RPCMEM_HEAP_DEFAULT, len);
```

The second parameter, `RPCMEM_HEAP_DEFAULT`, means that the allocated memory will be cached. To allocate an uncached buffer instead, use the `RPCMEM_HEAP_UNCACHED` parameter:

```
rpcmem_alloc(0, RPCMEM_HEAP_UNCACHED, len);
```

The reason for this optimization is that all cached buffers are shared across multiple processing units, so the cache coherency mechanism must be in place. Therefore, if you are certain the HLOS will not modify this allocated buffer other than during HVX processing, mark the memory buffer as uncached so that all mechanisms and related latencies will be circumvented.

3.5.1 Mark buffers as non-coherent

NOTE: This section was added to this document revision.

For all chipsets where IO coherency is applied, an alternative to marking the ION buffers as uncached is to register the buffers as non-coherent. The following example shows how to implement non-coherency:

```
flags = ION_FLAG_CACHED | RPCMEM_HEAP_NONCOHERENT;
rpcmem_alloc(0, flags, len);
```

3.6 Allocate DSP local buffers on the DSP

A common pitfall when designing algorithms over the FastRPC framework is to allocate all memory over rpcmem functions, and then send them all to the aDSP. Instead, allocate only what must be shared with rpcmem functions, and allocate the rest of the memory locally.

In many aDSP algorithms, an intermediate buffer is required to store the data temporarily. Typically, those buffers are local to the aDSP, and the HLOS is not required to access those intermediate buffers. We strongly recommend allocating those data buffers inside the aDSP implementation, instead of allocating them via the ION allocator and then passing them to the aDSP.

For example, scratch_buf of size 1024 is accessed only on the aDSP.

Instead of calling rpcmem_alloc() in the HLOS (stub) program and then passing the buffer as an argument:

```
scratch_buf = rpcmem_alloc(0, RPCMEM_HEAP_DEFAULT, 1024);
```

Call malloc in the aDSP (skel) program, and then use scratch_buf in the aDSP program:

```
scratch_buf = malloc(1024);
```

Locally allocating the buffers on the aDSP prevents unnecessary FastRPC overhead.

3.7 Other examples

In addition to the `calculator_android.html` example, the Hexagon SDK has other FastRPC framework examples:

```
<SDK root>\docs\FastCV\Image Downscale.html
<SDK root>\docs\Camera Streaming\Examples.html
```

This document does not discuss these examples. Remember the following recommendations when using these examples:

- `..\FastCV\Image Downscale.html` – Be familiar with the concept and usage of the DSP Computer Vision (CV) function.
- `..\Camera Streaming\Examples.html` – Be familiar with the concept and usage of camera streaming functionalities.

4 FastRPC debugging

The following procedures assume that debugging is performed on Windows 7 setups. While most steps in this section apply to Linux setup as well, some steps do not apply to Linux.

CAUTION: Some Qualcomm utility applications do not yet work reliably on Windows 10 setups. This document will be updated when support for Windows 10 is complete.

4.1 Common debugging steps

The following steps are common in many debugging procedures described in Section 4.2.

4.1.1 Collect logcat logs and kernel logs

To collect adb logcat logs:

```
adb logcat
```

To collect adb logcat while printing it to a file:

```
adb logcat |tee <filename>.txt
```

To collect Android kernel messages while printing them to a file:

```
adb shell cat /proc/kmsg/ (|tee kernel_msg.txt)
```

4.1.2 Collect messages in QXDM Professional™ tool

If you have QXDM Professional (QXDM Pro)[2], use the QXDM Pro help document for information on collecting the messages. For your convenience, following are common keyboard shortcuts:

Shortcut	Description
F3: Message View	Displays all collected messages
Alt+I	Clear all messages and logs
Ctrl+I	Save all messages and logs to a file
Alt+A	Save all messages (but not logs) to a text file
Alt+S	(When the slide bar is at the bottom of the screen) Toggle automatic scrolling of messages

[2] Obtained from CreatePoint

4.1.3 Collect messages if QXDM Pro is not available

The Hexagon SDK provides other options for collection messages:

- mini-dm

  Refer to the documentation in:

  ```
  <HEXAGON_SDK_ROOT>\docs\Debugging_Message%20Logs.html#mini-dm
  ```

  NOTE: mini-dm works reliably only on Qualcomm MTPs. It is not guaranteed to work on any other customer devices.

- HLOS kernel
  - To enable kernel driver error logs:

    ```
    adb shell dmesg
    ```

  - In `kernel/drivers/char/adsprpc_shared.h`:

    ```
    ifndef VERIFY_PRINT_ERROR
    #define VERIFY_EPRINTF(format, args) pr_err(format, args)
    #endif
    ```

- HLOS user space

  To capture logcat messages:

  ```
  adb logcat -s adsprpc
  ```

4.1.4 Enable crashes on the aDSP

A common problem for many customer devices is being unable to trigger or induce crashes on the aDSP. You must enable crashes if you want to trigger them.

The reason why crashes are sometimes not enabled is because Qualcomm enables crash-isolation protection. Enabling crashes is equivalent to disabling crash-isolation.

4.1.4.1 Restart the aDSP subsystem

The first crash isolation is the aDSP subsystem restart (SSR). With this isolation, crashing in the aDSP does not cause the crash in the entire system, but causes only the aDSP to restart locally.

1. Find out which subsystem is the aDSP by checking the name of each subsystem:

   ```
   adb shell cat /sys/bus/msm_subsys/devices/subsys<i>/name
   ```

   Where `i` is 0, 1, 2, …

   The number of i might be different for different customer devices:

   ```
   /sys/bus/msm_subsys/devices/subsys2 # cat name
        adsp
   /sys/bus/msm_subsys/devices/subsys3 # cat name
        slpi
   ```

   In this example, subsys2 is the aDSP.

2. Disable SSR for the aDSP subsystem using the following command:

   ```
   adb shell "echo SYSTEM >
   /sys/bus/msm_subsys/devices/subsys2/restart_level"
   ```

4.1.4.2 Set the debug mode for the aDSP User PD

The aDSP User PD has two modes:

- Default mode – A crash in the User PD does not trigger a crash in the aDSP.
- Debug mode – A crash in the User PD triggers a crash in the aDSP.

Set the Debug mode for the User PD by calling `HAP_set_userpd_mode()` in the algorithm/skeleton implementation on the aDSP. For example, use a function such as an initialization routine.

1. To disable the aDSP SSR from the CPU (per Section 4.1.4.1), set the Debug mode:

   ```
   adb shell setprop fastrpc.process
   ```

   This property treats all User PDs as critical on the remote processor (aDSP).

2. Alternatively, treat only the User PD being debugged as critical by setting the ADSP_PROCESS_ATTRS environment variable to the program that is running:

   ```
   adb shell ADSP_PROCESS_ATTRS=1 /data/calculator 0 0 4
   ```

4.1.4.3 Induce crashes

Occasionally, you might want to induce a crash to gather logs and other system states at a certain point. If you do not want to modify the code, the following method allows you to induce a crash in adb:

```
>adb root
>adb wait-for-device
>adb shell "echo c > /proc/sysrq-trigger"
```

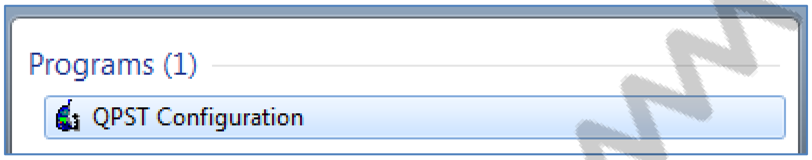
NOTE: There are many other ways to induce crashes. For simplicity, this document provides only this method.

4.1.5 Collect a crash dump

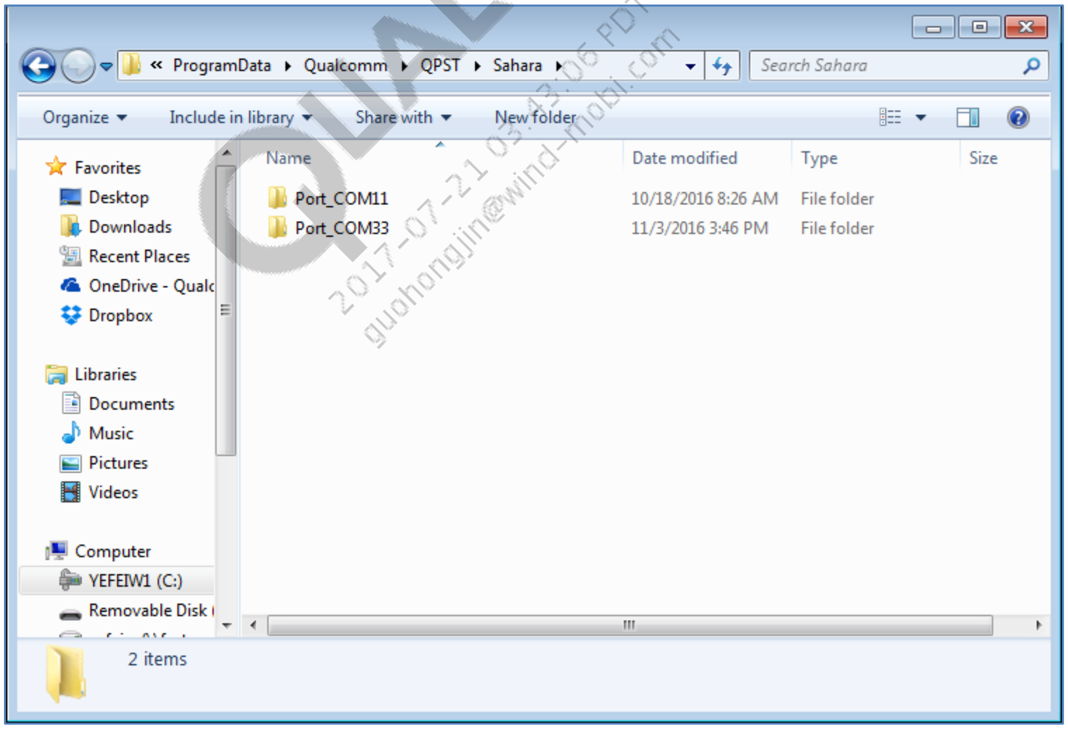
Ensure that the Qualcomm Product Support Tool (QPST) is enabled.

- If the QPST is enabled, the device automatically goes into Download mode, loads the crash dump onto the device, and reboots.
- If the QPST is not enabled, the device is stuck in Download mode while seeking the handshake signal to be sent from QPST. Loading will start as soon as QPST is enabled.

To enable the QPST, go to start menu and search for **QPST Configuration**.



After the crash dump is loaded, check `C:\ProgramData\Qualcomm\QPST\Sahara\` for the folders containing the loaded crash dumps.



Name	Date modified	Type	Size
Port_COM11	10/18/2016 8:26 AM	File folder	
Port_COM33	11/3/2016 3:46 PM	File folder	

As shown in this example, there might be more than one subfolder at that location. To determine which dump was loaded, either search by the COM port number from the device manager or determine the timestamps of the dumps. In this example, Port_COM33 is the recently loaded crash dump.

4.1.6 Load a DSP crash dump

When a crash occurs in the aDSP, refer to *Hexagon Multimedia: Android aDSP Crash Analysis* (80-NF768-29).

4.1.7 Measure latency in FastRPC calls

The major difficulty in measuring FastRPC latency is that the CPU and aDSP have different clock domains: 00:00:01.234 in the CPU clock does not equal to 00:00:01.234 in the aDSP clock.

We recommend measuring the overall latency of FastRPC as:

<CPU execution time> minus the <DSP execution time>

The following sections use the calculator example in the Hexagon SDK. You can further enhance the measurement by optimizing the functions to decrease the latency of the measurements themselves. You can also repeatedly execute the same FastRPC call many times, and print only the total timing values in the last iteration.

4.1.7.1 Profile the CPU execution time

On the CPU side, following these steps in `calculator_test.c`:

1. Add the following line to the file inclusion list:

```
#include <sys/time.h>
```

2. Add the following utility function:

```
unsigned long long get_time_msec()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return ((tv.tv_sec * 1000) + (tv.tv_usec / 1000));
}
//Note: It’s easy to convert the above function to measure in
microseconds.
```

3. Use the utility function after the call to the FastRPC functions, and subtract the timestamps to get the time elapsed:

```
uint64 timestamp = get_time_msec();
 calculator_sum(test, num, &result)
 timestamp = get_time_msec() — timestamp;
 printf(“- sum = %lld within %lld milliseconds\n”, result,timestamp);
```

4.1.7.2 Profile the DSP execution time

On the aDSP side, use the `HAP_perf_get_time_us()` function to get the execution time. In `calculator_imp.c`, follow these steps:

1. Add the following lines to the file inclusion list:

```
// profile DSP execution time (without RPC overhead) via HAP_perf api’s.
#include “HAP_perf.h”
```

2. Inside the `calculator_sum()` function, add the following lines:

```
Uint64 timestamp = HAP_perf_get_time_us();
FARF(HIGH, “execution time %lld”,HAP_perf_get_time_us() – timestamp);
```

   Take the first timestamp as the first line inside the function, and take the second timestamp as the last line before returning.

4.2 Debugging procedures

Following are debugging procedures for the common problems listed in Section 4.1. When new problems are introduced, Qualcomm will expand or renew these sections as necessary.

NOTE: Complete steps are provided for simpler problems. For more complicated problems, only the steps for loading the crash dump or generating more information are provided. More steps are required to find the root cause and resolve the problem.

4.2.1 FastRPC call fails but there are no crashes

When a FastRPC call fails without any crashes, verify that crashes are enabled as described in Section 4.1.4. If they are not enabled, perform the procedures in Section 4.1.4. to enable crashes, and then check for a crash.

If a crash still does not occur, collect logcat, kernel, and QXDM Pro logs to see what failed in the call. Following are typical failures.

4.2.1.1 Failures during FastRPC initialization

Kernel logs

Check whether the kernel logs have the following error, which indicates permission was denied when trying to open an adsprpc-smd device node:

```
01-01 07:04:48.150 4289 4289 W FastCVTest : type=1400 audit(0.0:77): avc:
denied { read } for name="adsprpc-smd" dev="tmpfs" ino=16418
scontext=u:r:untrusted_app:s0:c512,c768
tcontext=u:object_r:adsprpcd_device:s0 tclass=chr_file permissive=0
```

Add the name of the customer application in the sepolicy file to use this device. The existing content in the sepolicy file is a good example.

logcat

Check whether the logcat has following error, which indicates that an operation was not permitted and apps_dev_init failed:

```
01-01 00:06:56.645  2276  2315 E /system/vendor/bin/hbtp_daemon:
vendor/qcom/proprietary/adsprpc/src/fastrpc_apps_user.c:802:Error 57:
apps_dev_init failed. domain 2, errno "Operation not permitted"
```

Typically, this error occurs because the GLINK/SMD channel was not opened for the channel (aDSP/cDSP/mDSP). Check whether the DSP is up and running without any issues.

4.2.1.2 Signature failures

For signature failures, the QXDM Pro logs contain error messages that are like this example.

```
MSG          [08500/03]   QDSP6/Error                            00:03:59.021
sigverify.c  00553  209a::error: -1: -1 != (*num_segments =
GetProp_uint32(pHandle, “num_segments”,-1))
MSG          [08500/03]   QDSP6/Error                            00:03:59.022
sigverify.c  00623  209a::error: -1: 0 == Read_Hash_From_Devcfg(so_name, (const
byte**) &p_elf->pHashes, &p_elf->cbHashes, &num_segments)
MSG          [08500/02]   QDSP6/High                             00:03:59.022
sigverify.c  00568  209a:OEM ID ----------------------→ 0x0
MSG          [08500/02]   QDSP6/High                             00:03:59.022
sigverify.c  00569  209a:Debug Fuse Enabled ---------→ Yes
MSG          [08500/02]   QDSP6/High                             00:03:59.022
sigverify.c  00570  209a:Testsig Enabled ------------→ No
MSG          [08500/02]   QDSP6/High                             00:03:59.022
sigverify.c  00571  209a:Testsig file found ---------→ No
MSG          [08500/02]   QDSP6/High                             00:03:59.022
sigverify.c  00576  209a:module: Module is signed ---→ No
MSG          [08500/02]   QDSP6/High                             00:03:59.022
sigverify.c  00581  209a:module: Static hash found --→ No
MSG          [08500/03]   QDSP6/Error                            00:03:59.022
map_object.c  00491  96:signature verify start failed for libcalculator_skel.so
MSG          [08500/03]   QDSP6/Error                            00:03:59.025
rtld.c  00727  3099:dlopen failed, libcalculator_skel.so
```

Android has different versions, so there are two possible locations for the shared objects. If you are not sure of your version, use the first command to see if your version is newer. Use the second command to see if your version is older.

1. `adb shell ls /system/vendor/lib/rfsa/adsp`
2. `adb shell ls /system/lib/rfsa/adsp`

For example, when using the second command to check for `libadsp_hvx_add_constant.so`, you see the following results:

```
adb shell ls /system/vendor/lib/rfsa/adsp
libAMF_hexagon_skel.so
libadsp_hvx_add_constant.so
libadsp_hvx_skel.so
libadsp_hvx_stats.so
...
us-syncproximity.so
```

These results mean `libadsp_hvx_add_constant.so` was pushed into `/system/vendor/lib/rfsa/adsp`.

Check whether the signatures are generated and whether they are pushed into the right location:

1. Check whether the timestamp is correct. In this example, it is correct.

   ```
   adb shell ls /system/vendor/lib/rfsa/adsp —al
   -rw-r—r—root     root        28256 2016-07-16 09:04
   libAMF_hexagon_skel.so
   -rw-r—r—root     root        14736 2016-08-07 13:40
   libadsp_hvx_add_constant.so
   -rw-r—r—root     root       158136 2016-08-07 13:40 libadsp_hvx_skel.so
   ...
   -rw-r—r—root     root        78136 2016-07-16 09:04 us-syncproximity.so
   ```

2. To check whether the objects are built as debug, run the build command to build your shared objects. This example shows that you are building the binaries as the debug version:

   ```
   make tree V=hexagon_debug_dynamic_toolv72_v60
   ```

   In contrast, the following command builds the binaries as the release version, and you must sign the shared objects:

   ```
   make tree V=hexagon_Release_dynamic_toolv72_v60
   ```

3. To verify whether the signatures are present in the build, use the following command and search for the keyword, **testsig**.

   ```
   adb shell ls /system/vendor/lib/rfsa/adsp |grep testsig
   testsig-0x8fea24ce.so
   ```

   NOTE: With Hexagon SDK 3.0 installed, the test signature automatic loading script is located at `<sdk root>\scripts\testsig.py`.

4. Edit the Python file to modify the location where test signatures are pushed:

   ```
   cat ..\..\..\scripts\testsig.py |grep "/system"
          os.system('adb shell mkdir /system/lib/rfsa')
          os.system('adb shell mkdir /system/lib/rfsa/adsp')
          os.system('adb push '+testsig+' /system/lib/rfsa/adsp/')
   ```

4.2.1.3 User PD crashes

If there are crashes from the User PD, the QXMD Pro log contains messages like this example:

```
MSG          [08500/03]   QDSP6/Error                             00:02:26.278
fastrpc_port.c  00079  failed to enqueue msg 3840 3840 9310d530 1020100 A800A8000
4096
MSG          [08500/03]   QDSP6/Error                             00:02:26.280
fastrpc_port.c  00079  failed to enqueue msg 3840 3842 3 4020200 A80004000 8192
```

Because the User PD has already crashed, there is no skeleton function to which the call in the stub function will map. Therefore, the immediate symptom of a crash in User PD is a failure to enqueue FastRPC call messages.

The log messages are retrieved by removing the skeleton function implementation in the calculator examples (calculator_sum function in calculator_imp.c). If a FastRPC function only has a stub implementation but not a skel, User PD crashes occur.

If User PD crash messages are in the log, the crash was not enabled; otherwise, the crash in User PD brings down the entire system. To ensure that the crash is enabled, follow the steps in Section 4.1.4. Then you will have more information about the actual point of crash from the crash dump.

4.2.1.4 Calling functions on a crashed User PD

When calling functions and the User PD crashes, the QXDM Pro log lists the FastRPC error code 39:

```
MSG           [08500/03]   QDSP6/Error                         03:55:41.735
fastrpc_thread_queue.c  00771  012b::error: 39: !(nErr =
ftq_enqueue_on_group(procs, msg))
MSG           [08500/03]   QDSP6/Error                         03:55:41.735
fastrpc_port.c  00088  012b:failed to enqueue msg 1029 12441 e69095e0 0 00000000
4096
```

This message indicates two things:

- The FastRPC session was previously established, and it has crashed by the time we call this function.

  This error typically occurs when the applications processor repeatedly calls the same FastRPC function across many iterations, and User PD crashes occurred before the latest iteration calling the same function.

- The User PD crash did not bring down the entire system.

  Enable crashes (per Section 4.1.4) to find the actual point of the crash. Gather the logs and crash dumps accordingly.

Important FastRPC error codes:

```
#define  AEE_ENOSUCH              39 // no such name/port/socket/service
exists or valid name/port/socket/service exists or valid
#define  AEE_EOUTOFHANDLES        45 // out of handles
#define  AEE_ECONNRESET          104 // Connection reset by peer
```

4.2.2 Crashes in FastRPC functions

Following are some common problems when crashes occurred in FastRPC functions. When calling FastRPC functions, crashes can occur whether a User PD crash is enabled or not.

4.2.2.1 Crashes occur without enabling a User PD crash

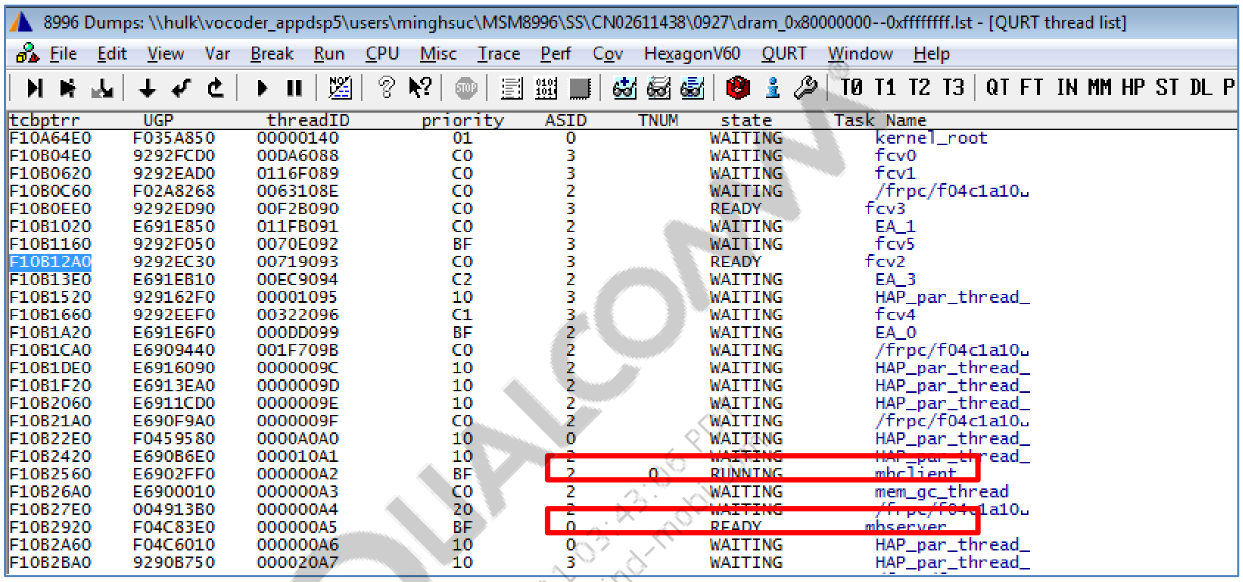
When crashes are not enabled, but there are still crashes when calling FastRPC functions, load the dump to check whether the FastRPC threads were running at the time of the crash. Then determine whether the crash is immediately caused by FastRPC functions in the rootPD.

To determine if a thread is from rootPD, follow these steps:

1. Load the crash dump.
2. In the Trace32 window, click the **QT** button.

3. Find the ASID number for the thread.
   - If the ASID number is 0, the thread is from rootPD.
   - Otherwise, the thread is from User PD.

   As shown in this example, the **mbserver** thread is on rootPD, and the running **mbclient** thread is on User PD.


8996 Dumps: \\hulk\vocoder_appdsp5\users\minghsuc\MSM8996\SS\CN02611438\0927\dram_0x80000000--0xffffffff.lst - [QURT thread list]
File Edit View Var Break Run CPU Misc Trace Perf Cov HexagonV60 QURT Window Help
T0 T1 T2 T3 QT FT IN MM HP ST DL P

```
tcbptrr     UGP         threadID        priority  ASID    TNUM    state      Task Name
F10A64E0    F035A850    00000140        01        0               WAITING      kernel_root
F10B04E0    9292FCD0    00DA6088        C0        3               WAITING      fcv0
F10B0620    9292EAD0    0116F089        C0        3               WAITING      fcv1
F10B0C60    F02A8268    0063108E        C0        2               WAITING      /frpc/f04c1a10.
F10B0EE0    9292ED90    00F2B090        C0        3               READY       fcv3
F10B1020    E691E850    011FB091        C0        2               WAITING      EA_1
F10B1160    9292F050    0070E092        BF        3               WAITING      fcv5
F10B12A0    9292EC30    00719093        C0        3               READY       fcv2
F10B13E0    E691EB10    00EC9094        C2        2               WAITING      EA_3
F10B1520    929162F0    00001095        10        3               WAITING      HAP_par_thread_
F10B1660    9292EEF0    00322096        C1        3               WAITING      fcv4
F10B1A20    E691E6F0    000DD099        BF        2               WAITING      EA_0
F10B1CA0    E6909440    001F709B        C0        2               WAITING      /frpc/f04c1a10.
F10B1DE0    E6916090    0000009C        10        2               WAITING      HAP_par_thread_
F10B1F20    E6913EA0    0000009D        10        2               WAITING      HAP_par_thread_
F10B2060    E6911CD0    0000009E        10        2               WAITING      HAP_par_thread_
F10B21A0    E690F9A0    0000009F        C0        2               WAITING      /frpc/f04c1a10.
F10B22E0    F0459580    0000A0A0        10        0               WAITING      HAP_par_thread_
F10B2420    E690B6E0    000010A1        10        2               WAITING      HAP_par_thread_
F10B2560    E6902FF0    000000A2        BF        2       0       RUNNING      mbclient
F10B26A0    E6900010    000000A3        C0        2               WAITING      mem_gc_thread
F10B27E0    004913B0    000000A4        20        2               WAITING      /frpc/f04c1a10.
F10B2920    F04C83E0    000000A5        BF        0               READY       mbserver
F10B2A60    F04C6010    000000A6        10        0               WAITING      HAP_par_thread_
F10B2BA0    9290B750    000020A7        10        3               WAITING      HAP_par_thread_
```

4.2.2.2 Crashes occur after enabling a User PD Crash

To load the crash dump and analyze the call stacks, internal states, and other information in the crash dumps, follow the steps in Section 4.1.5.

Check and resolve the following questions:

1. Was the customer thread running when the crash happened?
   - If yes, typically something is wrong inside the user algorithm.
   - If not, there might be system issues.
2. Is more than one User PD program running at the time of the crash, implying possible concurrency problems?
3. Are there many threads in the READY state, implying possible deadlocks or thread starvation?

4.2.3 No functional failures in FastRPC functions, but they are slow

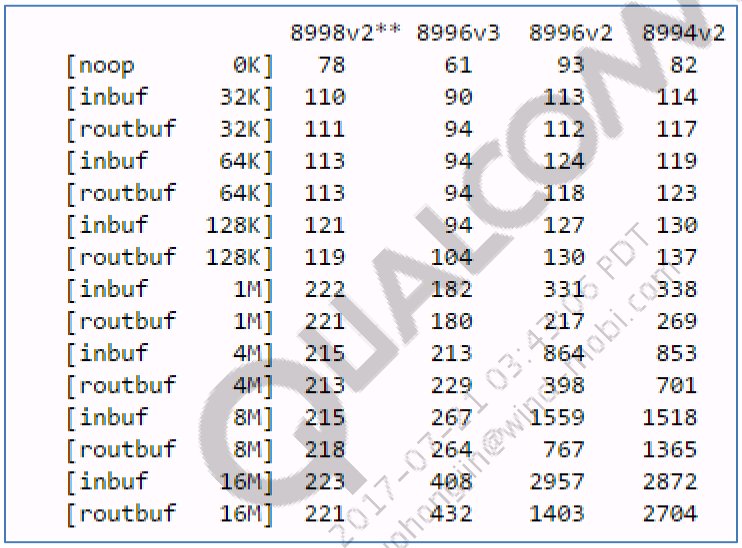
The following sections describe how to measure latency for FastRPC calls (FastRPC overhead) accurately.

4.2.3.1 Confirm FastRPC function latency

Hexagon SDK 3.1 and later versions have a FastRPC test function called **rpcperf**. This utility allows you to compare the latency vs. the following Qualcomm-measured table.

For more information on using the rpcperf tool, refer to Hexagon SDK documentation in:

```
<Hexagon_DSK_ROOT>\docs\Examples_Performance.html
```



```
                    8998v2** 8996v3  8996v2  8994v2
[noop         0K]   78        61      93      82
[inbuf       32K]  110        90     113     114
[routbuf     32K]  111        94     112     117
[inbuf       64K]  113        94     124     119
[routbuf     64K]  113        94     118     123
[inbuf      128K]  121        94     127     130
[routbuf    128K]  119       104     130     137
[inbuf        1M]  222       182     331     338
[routbuf      1M]  221       180     217     269
[inbuf        4M]  215       213     864     853
[routbuf      4M]  213       229     398     701
[inbuf        8M]  215       267    1559    1518
[routbuf      8M]  218       264     767    1365
[inbuf       16M]  223       408    2957    2872
[routbuf     16M]  221       432    1403    2704
```

Follow the steps in Section 4.1.7 to profile the total FastRPC latency, and then compare your results with this table.

NOTE: The values in the table are for ideal situations with minimum processing. We expect the actual values expected to be slightly higher.

4.2.3.2 Analyze FastRPC latency

If the latency is higher than expected, analyze the FastRPC latency:

1. Identify the possible reasons for latency (see the list of reasons).
2. Make the changes for each reason.
3. Profile FastRPC latency again to see if performance improves. If not, start over from step 1.

Common reasons for higher FastRPC latency

- If rpcperf numbers do not agree with the table in Section 4.2.3.1:
    - The boot image might not be in the performance kernel.

NOTE: For some Qualcomm builds, the default kernel is the debug kernel with extra code for debugging.

  To check for the image flavors, load `boot.img` and `system.img` from `android\out\target\product\msm8996\secondary-boot\`.

  - Clock voting might not be correct.

    Refer to the rpcperf documentation for information on how to vote for maximum clocks.

- If rpcperf numbers agree with the table in Section 4.2.3.1, but the latency of the actual user program is too high:
  - The total number of buffers being allocated via rpcmem operations is too high (more than dozens of MBs).
  - The total amount of memory passed via the FastRPC call is too high.
  - The user program unnecessarily uses cached buffers (Section 3.5).
  - The user program unnecessary uses intermediate buffers (Section 3.6).

A References

A.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Hexagon Multimedia: Fast RPC and Dynamic Loading User Guide for ADSP.BF.2.2 and 2.4	80-NF769-32
Hexagon Access Elite CAPIv2 API Interface Specification	80-N8098-1
Hexagon Multimedia: aDSP Firmware Overview for ADSP.BF.2.x	80-NF768-21
Sectools: Elfsigner/SecImage Tool User Guide	80-NM248-4
Enabling Secure Boot in MSM8996 Chipsets	80-NV396-81
QACT v6.x.x User Guide	80-VM407-9
Shared Memory Driver API Reference Guide	80-N1924-1
Hexagon Multimedia: Android aDSP Crash Analysis	80-NF768-29
Hexagon SDK 3.0 or later Hexagon600_SDK.WIN.3.0 Installer or the installer for a later version	

A.2 Acronyms and terms

Acronym or term	Definition
aDSP	Audio DSP
cDSP	Compute DSP
CV	Computer Vision
HLOS	High-level operating system
mDSP	Modem DSP
MPU	Memory protection unit
PD	Protection domain
QPST	Qualcomm Product Support Tool
RPC	Remote Procedure Call
sDSP	Sensors DSP
SMD	Shared Memory Driver
SMMU	System Memory Management Unit
SoC	System-on-chip
SSR	Subsystem restart
TCB	Task control block
TLB	Translation lookahead buffer